Linguagens de programação e estrutura de dados

unopar

Linguagens de programação e estruturas de dados

Gisele Alves Santana
Nathalia dos Santos Silva
Merris Mozer

Dados Internacionais de Catalogação na Publicação (CIP)

Santana, Gisele Alves
S2321 Linguagens de programação e estruturas de dados / Gisele Alves Santana, Nathalia dos Santos Silva, Merris Mozer – Londrina: Editora e Distribuidora Educacional S.A., 2018.
160 p.

ISBN 978-85-522-0314-8

1. Linguagem de programação. 2. Tipos abstratos de dados. I. Silva, Nathalia dos Santos. II. Mozer, Merris. II. Título.

CDD 001.6424

2018
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: http://www.kroton.com.br/

Sumário

Apresentação

Olá, aluno, seja bem-vindo!

A estrutura de dados é muito utilizada em diversas áreas para a resolução de problemas computacionais. A maneira de organização dos dados afeta diretamente a eficiência de um algoritmo. Dessa forma, dados que possuem melhor organização tendem a prover maior eficiência e rapidez aos algoritmos que manipulam esses dados. Para o entendimento das estruturas é necessário que você se lembre, principalmente, do conceito de algoritmos, assim como conheça os tipos de dados mais utilizados para a implementação de programas. Neste livro, a Linguagem C é adotada para a implementação das estruturas de dados por possuir alta flexibilidade e portabilidade.

Na Unidade 1, você será levado a entender a importância da estrutura de dados, assim como os principais tipos de estruturas utilizadas para a organização dessas informações; a aprender o que são funções, compreendendo a diferença entre passagem por valor e passagem por referência; a estudar sobre ponteiros, que são amplamente utilizados para a alocação dinâmica de memória, assim como compreender a utilização de registros para o armazenamento de informações que possuem dados de tipos diferentes. Para concluir esta unidade, você aprenderá o conceito de recursividade, que é utilizada principalmente para a resolução de problemas matemáticos mais complexos.

A Unidade 2 apresenta os tipos de dados pertinentes ao desenvolvimento de sistemas computacionais, assim como suas características, para que você possa tomar boas decisões quanto à escolha e definição da estrutura de dados a ser utilizada em seus projetos. Você irá aprender sobre os tipos de dados que podem ser implementados na linguagem C, incluindo comandos, funções e a sintaxe dessas implementações. Ao conhecer os tipos de dados abstratos e compostos, homogêneos e heterogêneos, você será capaz de criar estruturas adequadas para os seus programas.

O foco da Unidade 3 são as listas lineares e seus tipos principais: pilha e fila. Você irá entender a lógica utilizada para a implementação dessas estruturas de dados, utilizando técnicas de programação, como ponteiros e alocação dinâmica de memória, para sua implementação. Você também irá conhecer os principais tipos de algoritmos para a

pesquisa em estrutura de dados, bem como conceitos relacionados à classificação ou ordenação de elementos em uma estrutura.

Na Unidade 4 você aprenderá sobre dois tipos de estruturas de dados muito utilizados na área da computação: grafos e árvores. O foco está concentrado nas árvores binárias de busca. Dessa maneira, você irá conhecer as maneiras para a implementação desse tipo de árvore, assim como exemplos, simulações envolvendo as operações mais importantes e trechos de código na Linguagem C que implementam essas operações. Para finalizar, será definido o conceito de percurso ou travessia de uma árvore binária de busca, apresentando e ilustrando quatro tipos de percursos.

Além de todos os conceitos e exemplos apresentados neste livro, o material de estudos ainda contribuirá para que você possa treinar os conhecimentos adquiridos por meio da realização de exercícios orientados, Desejamos a você bons estudos e dedicação para a conclusão desta etapa.

Algoritmos e seus tipos de representação e estrutura de dados

Gisele Alves Santana

Objetivos de aprendizagem

Nesta unidade, você será levado(a) a entender a importância da estrutura de dados, assim como os principais tipos de estruturas utilizadas para a sua organização. Dentre os principais objetivos desta unidade, estão:

- Aprender o que são funções – conceito praticamente indispensável para a implementação de estruturas de dados – por meio da diferença entre passagem por valor e passagem por referência;
- Estudar sobre ponteiros, que são amplamente utilizados para a alocação dinâmica de memória;
- Compreender a utilização de registros para o armazenamento de informações que possuem dados de diferentes tipos;
- Aprender o conceito de recursividade, que é utilizada principalmente para a resolução de problemas matemáticos mais complexos.

Seção 1 | Tipos abstratos de dados e funções

Nesta seção você estudará os principais conceitos relacionados às Estruturas de Dados, conhecendo seus tipos mais importantes e associando-os a situações do seu cotidiano; entenderá o conceito de funções por meio de sua sintaxe, seus comandos básicos, as diferenças entre os tipos de passagem, de parâmetros, além de vários exemplos na Linguagem C de aplicação de funções.

Seção 2 | Técnicas de programação para a implementação de estruturas de dados

Nesta seção, você vai aprender sobre ponteiros, que possuem como conteúdo o endereço de memória de outra variável; ser apresentado(a) à alocação dinâmica de memória por meio de demonstrações de funções para a alocação e liberação de memória de um computador; estudar os registros, que possuem a capacidade de armazenar coleções de dados de diferentes tipos e, por fim, aprender sobre a recursividade, destacada como uma ferramenta de programação muito poderosa na resolução de problemas computacionais complexos.

Introdução à unidade

Nesta unidade, serão apresentados os conceitos básicos sobre Estrutura de Dados que, basicamente, definem os mecanismos para a sua organização, assim como os métodos de acesso aos dados processados por um programa.

Essa estrutura é muito utilizada em diversas áreas para a resolução de problemas computacionais. A maneira de organização dos dados afeta diretamente a eficiência de um algoritmo e, dessa forma, dados que possuem melhor organização tendem a prover maior eficiência e rapidez aos algoritmos que manipulam esses dados.

Para o entendimento das estruturas de dados, é necessário que você se lembre, principalmente, do conceito de algoritmos, assim como conheça os tipos de dados mais utilizados para a implementação de programas, e, nesta unidade, a Linguagem C, por possuir alta flexibilidade e portabilidade, será adotada para essa implementação.

Existem vários tipos de estruturas de dados e, dependendo do problema, uma determinada estrutura é mais adequada para a sua resolução do que outra; logo, ao final do estudo, espera-se que você conheça as características de alguns tipos e consiga identificar os mais adequados para a resolução de problemas específicos.

Para a implementação da maioria de tais estruturas são utilizadas funções para a manipulação de seus dados. Assim, esta unidade traz os conceitos mais importantes relacionados às funções, bem como vários exemplos em linguagem C. Algumas técnicas de programação essenciais para a implementação de estrutura de dados também são apresentadas, como a alocação dinâmica de memória, ponteiros e registros.

Para finalizar, esta unidade apresenta o conceito de recursividade, uma prática muito utilizada para a resolução de problemas computacionais complexos que, para ser melhor compreendida, seu conceito será exemplificado por meio de uma função matemática.

Seção 1

Tipos abstratos de dados e funções

Introdução à seção

Esta seção apresentará e ilustrará os principais conceitos relacionados às Estruturas de Dados, seus tipos mais importantes e, por fim, associará cada um deles com situações do seu cotidiano; ela também apresentará o conceito de funções por meio de sua sintaxe, seus comandos básicos, a diferença entre os tipos de passagem de parâmetros e vários exemplos na Linguagem C de aplicação de funções.

1.1 Introdução à estrutura de dados

Em programação, os tipos de dados definem o conjunto de valores que uma variável pode assumir ou as operações que podem ser realizadas sobre ela. Por exemplo, uma variável booleana (tipo lógico) pode assumir dois valores específicos: verdadeiro ou falso.

Ao declarar uma variável, automaticamente é reservada uma quantidade específica de bytes na memória para o armazenamento dos valores dessa variável. Assim, pode-se dizer que os tipos de dados são métodos para interpretar o conteúdo da memória de um computador.

Existem dois tipos de alocação de memória:

- Alocação estática: quando é alocado um espaço fixo e contíguo na memória para a variável;
- Alocação dinâmica: quando é alocado um espaço variável, que é criado segundo a necessidade do programa.

Um item especificado em termos das operações que pode ser realizado sobre ele é chamado de Tipo Abstrato de Dados (TAD). Vamos supor que precisamos projetar um item para realizar algumas tarefas; para isso, devemos especificar esse item de acordo com as operações realizadas, ao invés de sua estrutura interna.

Para entender melhor esse conceito, considere os passos para o projeto de um automóvel. Inicialmente, sabemos que todos os automóveis possuem características similares, como: pneus, volante, câmbio, motor etc., e ao ser analisado por esse aspecto, pode ser

considerado um tipo abstrato de dados, porém, para a construção do automóvel com as características especificadas é necessário decidir quais estruturas serão utilizadas para que esse trabalho tenha sucesso. Automóveis diferentes possuem estruturas diferentes, como: tipo de câmbio, tipo de motor, tipo de material etc., e através dessas estruturas pode-se construir o automóvel especificado; no entanto, ao extrairmos os detalhes da construção, todos os automóveis possuem as mesmas características. Logo, pode-se dizer que um tipo de dado representa uma descrição lógica, enquanto uma estrutura de dados representa uma descrição concreta.

O TAD (Tipo Abstrato de Dados) é o nível lógico e a estrutura de dados é o nível de implementação. Assim, a Estrutura de Dados é um método particular de se implementar um TAD.

Uma estrutura é construída dos tipos primitivos (inteiro, real, char etc.) ou dos tipos compostos (array, registro, etc.) de uma linguagem de programação.

Como exemplos de estrutura de dados, pode-se citar:

- Vetores (*arrays*).
- Registros (*structs*).
- Listas Ordenadas.
- Pilhas.
- Filas.
- Deques.
- Árvores.
- Grafos.

Essas estruturas de dados permitem que diversas operações sejam realizadas. Entre as mais utilizadas, destacam-se:

- Criação (declaração).
- Percurso.
- Busca.
- Inserção.
- Alteração.
- Exclusão.

Independentemente do tipo de dado com o qual se deseja trabalhar, primeiramente é realizada a operação de criação; em seguida, pode-

se realizar inclusões, alterações ou remoções de dados. Outro tipo de operação que pode ser realizada é o percurso, que faz a varredura de todos os elementos armazenados em uma estrutura de dados.

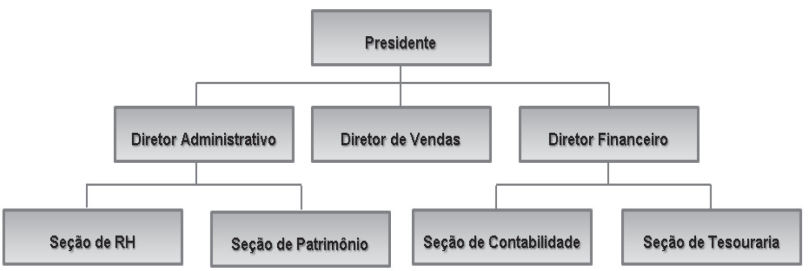
1.2 Estrutura de Dados no Dia a dia

Quando se realiza um cadastro de clientes, quais são os dados mais importantes que devem ser considerados, por exemplo: a idade ou a cor dos cabelos? Bem, isso vai depender muito dos requisitos levantados na elaboração do projeto do sistema. Além disso, deve-se considerar as operações que serão necessárias para a manipulação dos dados, por exemplo: como encontrar um cliente ou inserir um novo cliente?

Como foi visto, a estrutura de dados é utilizada, principalmente, para a organização das informações, proporcionando rapidez no momento da recuperação de algum item; no entanto, como as estruturas de dados podem ser aplicadas no dia a dia?

Imagine a organização de uma empresa que possui um presidente, um diretor administrativo (com as seções de recursos humanos), um diretor de vendas e um diretor financeiro (com as seções de contabilidade e tesouraria). Geralmente, essa hierarquia é representada graficamente por um organograma, que pode ser associado à estrutura de uma árvore, conforme observado na Figura 1.1.

Figura 1.1 | Modelo de árvore



Fonte: elaborada pela autora.

Outro tipo de associação das estruturas de dados com eventos do cotidiano pode ser feita em relação às caixas de pizzas, que geralmente são empilhadas pelo entregador, conforme Figura 1.2. Essa estrutura pode ser associada a uma pilha.

Figura 1.2 | Modelo de pilha

Fonte: elaborada pela autora.

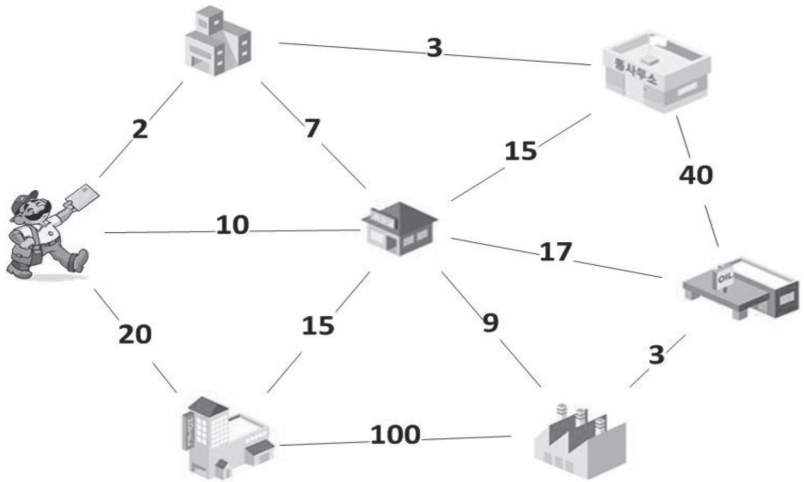
Você já notou a posição na qual as pessoas esperam sua vez por atendimento em um banco? Elas geralmente formam uma fila, por ordem de chegada, conforme Figura 1.3. Assim como a fila da vida real, na estrutura de dados a fila também tem as mesmas características.

Figura 1.3 | Modelo de fila

Fonte: <https://imgs.jusbr.com/publications/noticias/images/625121495215303.jpg>. Acesso em: 12 ago. 2017.

O grafo é outro tipo de estrutura de dados que pode ser associado com situações cotidianas. Os possíveis trajetos de um carteiro podem ser representados através desse tipo de estrutura de dados, conforme observado na Figura 1.4. Geralmente, os grafos possuem um peso associado a cada aresta, que também pode ser entendido como o custo gasto para o percurso de um vértice a outro.

Figura 1.4 | Modelo de grafo



Fonte: <http://4.bp.blogspot.com/-j9h8d0dzM2I/UbSAXvd3nII/AAAAAAAAACM/0wqWPDB1_Ys/s1600/Desenho1.png>. Acesso em: 12 ago. 2017.

Questão para reflexão

Você conhece o sistema de diretórios utilizado pelo sistema operacional Windows? Qual o tipo de estrutura de dados que você acha que é empregado para a organização dos dados (pastas e arquivos) do seu computador?

De acordo com Mizrahi (2006), existem dois tipos de estruturas de dados: lineares e não lineares. Nas estruturas lineares, o primeiro e o último elemento são bem definidos e os elementos intermediários possuem um antecessor e um sucessor. Exemplos de estruturas lineares: filas, pilhas, vetores etc. Já nas estruturas não lineares existe uma relação hierárquica ou qualquer outro tipo de relação entre os elementos, e o mais importante é saber identificar a melhor estrutura para a resolução de cada tipo de problema.

Segundo Tenenbaum, Langsam e Augenstein (2004), a manipulação dos dados em uma estrutura pode ser feita de forma sequencial ou encadeada. Na forma sequencial, o espaço de memória é pré-alocado no momento em que a estrutura é definida, tendo, assim, um tamanho fixo; já na forma encadeada, o tamanho alocado é inicialmente

desconhecido, sendo o espaço reservado conforme a necessidade e em tempo de execução.

Neste livro, nós veremos primeiramente as estruturas de dados que são manipuladas de forma sequencial, como os vetores, por exemplo; mas, antes de iniciarmos esse estudo é fundamental saber como manipular as informações de uma estrutura de dados.

Imagine uma lista de notas e as operações que se pode realizar com ela. Essa lista inicialmente está vazia, mas algumas notas serão inseridas, depois removidas, alteradas, e até mesmo uma operação de busca por uma determinada nota poderá ser realizada. Toda vez que se desejar inserir uma nova nota na lista, um mesmo conjunto de instruções será executado. Nesse caso, as boas práticas de programação sugerem que se crie uma função chamada "inserir", por exemplo, e sempre que precisarmos inserir uma nota, basta solicitar que o computador execute a função "inserir".

Em programação, todas essas operações para a manipulação de uma estrutura de dados são implementadas por meio de funções, e a linguagem de programação mais utilizada para isso é a linguagem C, uma vez que possui alta flexibilidade e portabilidade. Assim, nesta unidade, estudaremos algumas técnicas importantes dessa linguagem para a implementação das estruturas de dados.

Para saber mais

O C é uma linguagem de propósito geral, sendo adequada à programação estruturada. No entanto, é mais utilizada para escrever compiladores, analisadores léxicos, bancos de dados, editores de texto etc. A linguagem C pertence a uma família de linguagens cujas características são: portabilidade, modularidade, compilação separada, recursos de baixo nível, geração de código eficiente, confiabilidade, regularidade, simplicidade e facilidade de uso. Nos links a seguir, você tem acesso a apostilas com vários conceitos dessa linguagem de programação:

Disponível em: <ftp://ftp.unicamp.br/pub/apoio/treinamentos/linguagens/c.pdf>. Acesso em: 12 ago. 2017.

Disponível em: http://www2.dcc.ufmg.br/disciplinas/pc/source/introducao_c_renatocm_deeufmg.pdf> . Acesso em: 12 ago. 2017.

1.3 Funções

"Uma função é um conjunto de instruções desenhadas para cumprir uma tarefa particular, agrupadas numa unidade com um nome para referenciá-la" (MIZRAHI, 2006, p. 117). As funções são usadas para criar pequenos pedaços de códigos separados do programa principal e servem para agrupar um conjunto de instruções de acordo com a tarefa que elas desempenham. A principal finalidade das funções é impedir que o programador tenha que escrever o mesmo código repetidas vezes. Para exemplificar o conceito de função, imagine um sistema de controle de estoques; nesse sistema, algumas operações como inclusão ou exclusão de produtos são executadas com certa frequência. Dessa maneira, essas operações podem ser implementadas em forma de funções e todas as vezes que houver a necessidade de cadastrar um novo produto, por exemplo, a função de "inclusão" é chamada.

1.3.1 Funcionamento

As funções agrupam um conjunto de comandos e associam a ele um nome, e o uso desse nome é uma chamada da função. Após sua execução, o programa volta ao ponto do programa principal situado imediatamente após a chamada, e a essa volta damos nome de retorno.

1.3.2 Sintaxe

"A sintaxe pode ser definida pelo conjunto de regras que definem as sequências corretas dos elementos de uma linguagem de programação" (MIZRAHI, 2006, p. 117). A sintaxe de uma função é muito semelhante a de uma função main( ). A única diferença é que a main( ) possui um nome especial, pois essa função é a primeira a ser chamada quando o programa é executado.

Os programas em linguagem C podem ser executados em diversos compiladores gratuitos, incluindo o Code Blocks (disponível para download em: http://www.codeblocks.org/) ou Dev C++ (disponível para download em: http://www.bloodshed.net/devcpp.html).

A seguir, tem-se um exemplo da estrutura de uma função:

```
<tipo> <nome>(<parâmetros>)
{
    <declarações locais>;
    <comandos>;
     return //expressão ou valor compatível com o tipo de retorno
}
```

Como se percebe, os elementos básicos de uma função são: tipo, nome e parâmetros. O "tipo" define o tipo de dado que a função retornará como o resultado de sua execução; o "nome" indica qual é o nome da função, já os parâmetros são utilizados para transmitir informações para a função.

1.3.3 Chamando as Funções

Uma chamada a uma função é feita escrevendo-se o nome dela seguido dos parâmetros fornecidos (entre parênteses). Se não houver parâmetros, ainda assim, devem ser mantidos os parênteses, para que o compilador diferencie a chamada de uma função e a de uma variável. Além do mais, o comando de chamada da função deve ser seguido de ponto e vírgula, e as funções apenas podem ser chamadas depois de terem sido declaradas.

Para chamar à função "potencia", por exemplo, deve-se escrever a seguinte linha de instrução: potencia ( );

No próximo exemplo, tem-se o código de um programa que possui uma função chamada "mensagem". Essa função vem antes do programa principal e é chamada (invocada) dentro do mesmo. A função "mensagem" é do tipo *void*, ou seja, não retorna nenhum valor ao programa principal. Ao compilar esse código, a saída do programa será a frase: "Olá, eu sou uma função".

```
#include<iostream.h> // inclusão das bibliotecas

void mensagem ( ) // não retorna nenhum valor ao programa principal. Não há
ponto-e-vírgula aqui!
{
    printf ("Ola, eu sou uma função");
}

int main () {   //programa principal
   mensagem ( ); //chamando uma função sem argumentos
}
```

É possível notar, no exemplo a seguir, outro programa que possui uma função chamada "potencia", que calcula o quadrado de um determinado valor. Essa função é invocada no programa principal através do comando "potencia ();". A partir desse momento, a execução do programa principal para, e a função é iniciada, criando duas variáveis

do tipo inteiro. Em seguida, é solicitado que o usuário digite um valor, que será multiplicado por ele mesmo e o resultado será atribuído à variável "pot", que é escrita na tela. Ao final da execução de todos os comandos da função, a execução do programa retornará à linha posterior, à chamada da função no programa principal.

```
#include<iostream.h> // inclusão das bibliotecas

int potencia ( ) //não há ponto-e-vírgula aqui!
{
    int x, pot;
    printf ("Digite um numero: \n");
    scanf("%d", &x);
    pot = x * x;
    printf("\nPotencia = %d", pot);
}

int main ()
{
   potencia ( ); //chamando uma função sem argumentos
   return 0;
}
```

Geralmente, a maioria dos programas possui várias funções, cada uma executando uma tarefa específica.

Para saber mais

A Linguagem C possui uma biblioteca chamada: <math.h>. Nessa biblioteca, estão disponíveis várias funções matemáticas, como: potência, raiz quadrada etc.

Confira mais informações no link a seguir: <http://linguagemc.com.br/a-biblioteca-math-h/>. Acesso em: 12 ago. 2017.

1.3.4 Protótipos de Funções

Até o conteúdo estudado, todas as funções ficaram localizadas logo no início do programa e após as inclusões das bibliotecas, pois não se pode utilizar uma função sem antes declará-la. Porém, existe uma forma de se escrever uma função depois do programa principal e

isso é possível com a utilização de protótipos de funções.

"Os protótipos podem ser considerados como declarações de funções" (MIZRAHI, 2006, p. 120). O protótipo é colocado no início do programa (após a inclusão das bibliotecas), estabelecendo o tipo, nome e a lista de parâmetros da função.

Suponha que o programa contenha uma função chamada "potencia". A sintaxe do protótipo dessa função será:

int potencia (int a);

Repare que é igual ao cabeçalho da definição da função; porém, ao invés do { (abre chaves) tem-se o; (ponto e vírgula).

A seguir, será apresentado um programa completo que utiliza o recurso de protótipos de funções.

```
#include<iostream.h>

int potencia (); // protótipo da função

int main ()
{
   potencia ( ); //chamando uma função
   system("PAUSE);
   return 0;
}

int potencia ( ) // definição da função
{
    int x, pot;
    printf ("Digite um numero: \n");
    scanf("%d", &x);
    pot = x * x;
    printf("\nPotencia = %d", pot);
}
```

Para saber mais

No link a seguir você encontra a definição e utilidade do comando system("PAUSE"), que é basicamente usado para interromper a execução de um programa.

Disponível em: <http://www.ime.usp.br/~elo/IntroducaoComputacao/Esqueleto%20de%20um%20programa%20em%20C.htm>. Acesso em:

12 ago. 2017.

Já o comando return 0 pode ser utilizado quando a função não retorna nenhum valor. No link a seguir, você encontra mais explicações sobre esse comando:

Disponível em: <http://linguagemc.com.br/funcoes-em-c/>. Acesso em: 12 ago. 2017.

1.3.5 Comando Return

"O comando *return* termina a execução de uma função e retorna o controle para a instrução seguinte do código da chamada da função" (MIZRAHI, 2006, p. 123). Quando uma função não tem um tipo de retorno definido, o compilador considera que o tipo de retorno adotado é *void*.

Existem três sintaxes possíveis associadas ao comando *return*:

- return;
- return expressão;
- return (expressão).

Para ilustrar a utilização do comando *return* foi desenvolvida uma função chamada "Potencia", que recebe como parâmetro o valor da variável "x" do programa principal e retorna o quadrado desse valor.

```
int Potencia (int x)
{
    return x*x;
}
```

Uma questão importante em relação a esse comando é o fato do mesmo poder retornar apenas UM valor. Se o programa necessita que mais valores sejam modificados por uma função, outra maneira de passagem dos parâmetros se faz necessária. Nós estudaremos sobre esse assunto mais adiante nesta seção.

1.3.6 Variáveis locais e globais

Segundo Mizrahi (2006), um conceito muito importante em funções é o de variáveis locais. A declaração das variáveis da função

deve vir no início da função, antes de qualquer outro comando. Uma variável declarada dentro de uma função é "local", ou seja, só existe dentro da função. Ao ser iniciada a função, a variável é criada, e quando a função termina, a variável é apagada.

O escopo de uma variável é definido pelas regiões onde a variável pode ser utilizada. Por exemplo, as variáveis declaradas no início da função principal podem ser utilizadas em qualquer lugar dentro da função principal, porém, apenas DENTRO dela, ou seja, NÃO podem ser utilizadas em outra função.

Variáveis declaradas no mesmo escopo (mesma função) precisam ter nomes diferentes, mas nomes podem ser "reaproveitados" em outros escopos (outras funções).

1.3.7 Parâmetros das funções

"Os parâmetros de uma função são utilizados para transmitir informações para a função", (MIZRAHI, 2006, p. 125). Uma função pode receber qualquer número de argumentos, sendo possível escrever uma função que não receba nenhum argumento. No caso de uma função sem argumentos pode-se escrevê-la de duas maneiras:

- Deixando a lista de argumentos vazia (mantendo os parênteses);
- Colocando o tipo void entre os parênteses.

Os parâmetros são inseridos entre os parênteses após o nome da função e separados por vírgulas. A seguir, será apresentado o exemplo de uma função que calcula a área de um retângulo, dados os valores da base e da altura. Repare que no programa principal é declarada uma variável chamada "ret", que recebe o valor retornado pela função "área".

```
#include<iostream.h> // inclusão das bibliotecas

int area (int base,int altura); // protótipo da função

int main ()   { // programa principal
   int a,b,ret;
   printf("\nDigite o valor da base do retângulo: ");
   scanf("%d", &b);
   printf("\nDigite o valor da altura do retângulo: ");
   scanf("%d", &a);

   ret = area (b,a); //chamando a função com a passagem das variáveis b, a
   printf("\nArea do retângulo =  %d", ret);
   system ("PAUSE");
}

int area (int base, int altura) { // função
      int a;
      a = base * altura;
      return a; // retorna para o programa principal o valor da área calculada
 }
```

Se mais de um parâmetro for necessário, ou seja, passar mais de um valor para uma função, esses podem ser colocados na lista de parâmetros separados por vírgulas. Pode-se passar quantos parâmetros desejar. Existem duas formas utilizadas para passagem de parâmetros: passagem por valor e passagem por referência.

1.3.8 Passagem por Valor

"A passagem por valor é a forma mais comum utilizada para passagem de parâmetros" (MIZRAHI, 2006, p. 126). Por exemplo, considere funções trigonométricas, como seno, cosseno etc. A função seno recebe o valor de um ângulo (um número real) e devolve o seno desse ângulo. Vejamos:

float seno (float angulo);

Quando as variáveis são passadas por valor, a função cria novas variáveis do mesmo tipo e copia nelas os valores dos parâmetros (variáveis) passados. Assim, as funções não têm acesso às variáveis da função principal (int main), não podendo modificar os valores das mesmas.

1.3.9 Passagem por Referência

Até agora foi visto que as funções podem retornar apenas um único valor. Porém, algumas vezes, é necessário retornar mais de um, e quando isso acontece, utiliza-se a passagem por referência. A principal vantagem nesse tipo de passagem é que a função pode acessar as variáveis da função principal.

Para tanto, utiliza-se um operador chamado de operador unário de referência, que é simbolizado por "&". Esse operador cria outro nome para uma variável já criada. Considere as instruções:

```
int n;
int & n1 = n;
```

Analisando as instruções, são declaradas duas variáveis, "n" e "n1". O operador &n1 = n indica que agora "n1" é outro nome para "n". Ou seja, todas as operações em qualquer das duas variáveis têm o mesmo resultado. O operador "&" faz referência ao endereço de memória de uma variável; no entanto, uma referência não é uma cópia da variável a quem se refere, mas sim, a mesma variável sob diferentes nomes.

Analisemos agora o exemplo de uma função que tem o objetivo de alterar o valor de duas variáveis, conforme programa apresentado a seguir.

```
#include<iostream.h>

int altera (int x, int y) //Função
{
    x = 5;
    y = 15;
}

int main() {
   int a, b;
   a = 10;
   b = 20;
   printf("\nValor de a: %d", a); //Imprime na tela o valor da variável a
   printf("\nValor de b: %d", b); //Imprime na tela o valor da variável b
   altera (a,b); //Chamada da função
   printf("\nValor de a: %d", a);
   printf("\nValor de b: %d", b);
   system ("PAUSE");
}
```

Compilando o programa, nota-se que os valores das variáveis "a" e "b" não foram alterados no programa principal. Quando a função "altera" é chamada e inicializada, os valores são trocados (x = a e y = b), entretanto, quando a função termina, as variáveis da função (x e y) são destruídas.

Como fazer para que a função "altera" mude realmente os valores das variáveis "a" e "b"?

Nesse caso, utiliza-se o operador unário "&" antes do nome das variáveis que são passadas por parâmetro, conforme o cabeçalho da função:

int altera (int& x, int& y) {

A seguir será apresentado um programa que realmente faz a modificação dos valores das variáveis no programa principal.

```
#include<iostream.h>
int altera (int &x, int &y) //Função
{
    x = 5;
    y = 15;
}
int main()
{
   int a, b;
   a = 10;
   b = 20;
   printf("\nValor de a: %d ", a); //Imprime na tela o valor da variável a
   printf("\nValor de b: %d ", b); //Imprime na tela o valor da variável b
   altera (a,b); //Chamada da função
   printf("\nValor de a: %d", a);
   printf("\nValor de b: %d", b);
   system ("PAUSE");
}
```

O operador unário retorna o endereço de memória da variável e, com esse tipo de passagem, não se tem mais cópias das variáveis, mas sim, o acesso direto às variáveis da função principal. Então, alterados os valores das variáveis na função, altera-se também os seus valores no programa principal, pois nesse caso, estaremos lidando com as

mesmas variáveis.

E, então, conhecidos os conceitos mais importantes relacionados à utilização de funções, agora é o momento de aprendermos sobre um conceito muito utilizado para a manipulação de estruturas de dados: os ponteiros.

Finalizando a seção

Nesta seção, você aprendeu sobre os principais conceitos relacionados às Estruturas de Dados, assim como conheceu seus tipos mais importantes; foi apresentado(a) ao conceito de funções, à sua sintaxe, aos seus comandos básicos, à diferença entre os tipos de passagem de parâmetros e a vários exemplos na Linguagem C de aplicação de funções.

Atividades de aprendizagem

1. Você já adquiriu algumas habilidades de programação no decorrer desta seção. Dessa forma, é proposto que você utilize seus conhecimentos prévios relacionados aos algoritmos e à Linguagem de programação C e implemente um programa que receba do usuário (no programa principal) duas variáveis: x, y. Através de uma função, calcule a potência do valor x (x é a base, y é o expoente).

2. Você aprendeu, nesta seção, a diferença entre a passagem por valor e passagem por referência para a implementação de funções. Considerando o que foi estudado, faça um programa em C que contenha uma função que receba dois valores inteiros por parâmetro e retorne-os ordenados em ordem crescente.

Seção 2

Técnicas de programação para a implementação de estruturas de dados

Introdução à seção

Nesta seção, você irá aprender que os ponteiros possuem como conteúdo o endereço de memória de outra variável; será apresentado(a) à alocação dinâmica de memória, através de demonstrações de funções para a alocação e liberação de memória de um computador; aprenderá que os registros possuem a capacidade de armazenar coleções de dados de diferentes tipos e, por fim, obterá explicações a respeito da recursividade, destacando-se como uma ferramenta de programação muito poderosa e empregada para a resolução de problemas computacionais complexos, podendo ser usada sempre que for possível expressar a solução de um problema em função do próprio problema.

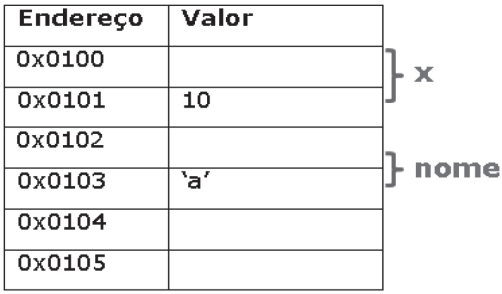
2.1 Ponteiros

"Ponteiros são variáveis cujo conteúdo é um endereço de memória" (MIZRAHI, 2006, p. 144). Assim, como um ponteiro endereça uma posição de memória que contém valores e um determinado endereço, diz-se que ele *aponta* para esse endereço de memória. Logo, como o valor do ponteiro é o endereço de outra variável, diz-se que ele *aponta* para essa variável. Na linguagem C, as variáveis estão associadas a um nome, um tipo, um valor e um endereço de memória. Por exemplo:

```
int x = 10;
char nome = "a";
```

Na memória, o armazenamento dessas variáveis é ilustrado na Figura 1.5.

Figura 1.5 | Armazenamento das variáveis na memória



Fonte: elaborada pelo autora.

A variável inteira "x" está armazenada no endereço "0x0100". Ela utiliza dois bytes de memória (quando um objeto usa mais de um byte, seu endereço é aquele onde ele começa - nesse caso, 0x0100 e não 0x0101). A variável do tipo char está armazenada no endereço "0x0103" e usa um byte de memória, e o compilador é responsável por controlar os locais de armazenamento das variáveis.

2.1.1 Operador de endereço (&)

O operador de endereço (&) fornece o endereço de memória onde está armazenada uma variável. Lê-se "*o endereço de*". Esse operador pode ser usado conforme nas expressões a seguir:

&x tem valor 0x0100

&nome tem valor 0x0103

2.1.2 Declaração de Ponteiros

Para declarar um ponteiro basta utilizar o operador *(asterisco) antes do nome da variável. Ponteiros são tipados, ou seja, devem ter seu tipo declarado e somente podem apontar para variáveis do mesmo tipo. Acompanhe os exemplos a seguir:

```
int *pont; // define um ponteiro para inteiro chamado pont.

float *nota; // define um ponteiro para real chamado nota.

char *sexo; // define um ponteiro para caractere chamado sexo.

struct aluno *faculdade; // define um ponteiro para uma estrutura chamado faculdade.
```

2.1.3 Inicialização de Ponteiros

Até agora os ponteiros foram declarados, mas ainda não foram

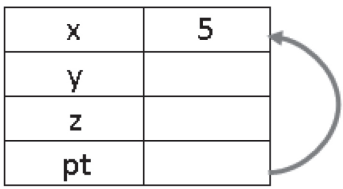
inicializados, ou seja, eles apontam para um lugar indefinido na memória, e é necessário, antes de ser utilizado, que ele seja apontado para algum lugar conhecido, ou, em outras palavras, é preciso que ele seja inicializado.

Como exemplo de inicialização de ponteiros, considere o seguinte trecho de código:

```
int x = 5;
int *pt;
pt = &x;
```

No exemplo, foi criada uma variável do tipo inteiro chamada "x" e atribuído o valor 5 para ela; bem como foi criado um ponteiro para o inteiro "pt". A linha de código: *pt = &x* significa que a expressão &x fornece o endereço de "x", o qual é armazenado em "pt", que passa a apontar para o endereço de memória da variável "x". Na Figura 1.6, tem-se a ilustração do funcionamento do ponteiro "pt".

Figura 1.6 | Funcionamento de um ponteiro



Fonte: elaborada pelo autora.

Pode-se alterar o valor de "x" utilizando "pt". Para isso, deve-se usar o operador "inverso" do operador **&**, que é o operador *.

O operador * possui dois empregos distintos no uso de ponteiros:

- Na declaração de uma variável, indicando que ela é um ponteiro;
- Na implementação do programa, sendo utilizado para a manipulação do **conteúdo ou valor** de variável.

O valor de "x" pode ser alterado através do ponteiro "pt", por exemplo: *pt = 15; //altera o valor de "cont" para 15.

No código a seguir, tem-se o exemplo de um programa na linguagem C que declara uma variável do tipo inteiro, assim como um ponteiro do mesmo tipo. Esse ponteiro é inicializado, ou apontado

para a variável "x". Ao compilar o programa, percebe-se que o valor do ponteiro é o endereço de memória da variável "x".

```
#include <iostream.h>

int main()
{
  int x;
  int *pt;  // declara um ponteiro para uma variável do tipo inteiro
  x = 10;
  pt = &x;
  printf("Valor de x: %d, Endereco de x: %d\n", x, &x);
  printf("Valor de pt: %d, Conteúdo de pt: %d\n", pt, *pt);
  system("PAUSE");
}
```

Questão para reflexão

Quais os principais problemas que podem ocorrer quando um ponteiro não é inicializado?

2.1.4 Impressão de Ponteiros

Na linguagem de programação C, pode-se imprimir o valor armazenado no ponteiro (um endereço) usando a função "printf" com o operador "%p" na string de formato.

A seguir será apresentado um exemplo de código com a impressão de ponteiros.

```
#include < {
  int a; iostream.h>
int main()

  int *pt; //declaração do ponteiro pt
  pt = &a; // pt aponta para a
  printf("O endereço de a é: %p\n", pt);
}
```

Os ponteiros também são utilizados para a alocação dinâmica de memória, muito utilizada para a implementação de estruturas de dados

mais complexas.

2.2 Alocação dinâmica de memória

Em C, pode-se alocar dinamicamente memória durante a execução de um programa, e tal alocação pode ser feita com a função *malloc*. Já, para a liberação de memória utiliza-se a função *free*, e no exemplo que se segue, pode-se observar o uso de ambas:

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

int main()
{
 int *pt;
 pt = (int*)malloc(sizeof(int));
 // Aloca memória necessária para um inteiro e coloca em 'pt' esse endereço.

  if (pt == NULL) //Se não existir memória disponível
  {
    printf("Memória insuficiente.");
  }
  printf("Endereço de pt: %p\n",  pt);
  *pt = 5;
  printf("Conteúdo de pt: %d\n", *pt);  // Imprime o valor 5
  free(pt);  // Libera a memória alocada para o ponteiro
  system("PAUSE");
```

Estudados os conceitos mais importantes relacionados à utilização de funções, ponteiros e alocação dinâmica de memória, agora é o momento de aprendermos sobre os registros, muito utilizados para a implementação de pilhas, filas ou árvores dinâmicas.

2.3 Registros

Na Unidade 2, o estudo será pautado em vetores, que são estruturas de dados homogêneas e que, basicamente, armazenam vários valores, porém, todos de um mesmo tipo. No entanto, o que fazer quando se tem coleções de dados que possuem tipos diferentes, como uma ficha de cadastro de clientes?

Uma ficha de cadastro pode possuir alguns campos, como:

Nome: *string*

Endereço: *string*

Telefone: *string*

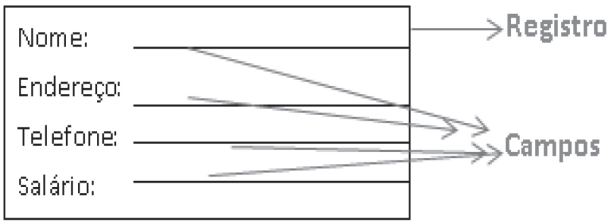
Salário: *float*

Idade: *int*

Para resolver esse problema existem os registros, que "são conjuntos de dados logicamente relacionados, mas que podem possuir tipos diferentes de variáveis, como: inteiro, real, string etc." (MIZRAHI, 2006, p.237). Um registro ou estrutura (*struct*) é um grupo de itens no qual cada item possui um identificador próprio, sendo cada um deles conhecido como um membro da estrutura. Um registro permite agrupar dados de diferentes tipos em uma mesma estrutura (ao contrário dos vetores que possuem elementos de um mesmo tipo).

Cada componente de um registro pode ser de um tipo diferente (*int, char etc.* Esses componentes são referenciados por um nome. Em várias linguagens de programação, uma estrutura é chamada "registro" e um membro é chamado de "campo", conforme Figura 1.7. Um campo é um conjunto de caracteres com o mesmo significado.

Figura 1.7 | Elementos de um registro



Fonte: Elaborada pelo autora (2017).

2.3.1 Declaração de um Registro

Geralmente, um registro ou struct é declarado após a inclusão das bibliotecas e antes do programa principal. A sintaxe básica da declaração de um registro é mostrada a seguir:

```
struct <identificador>
{
  <listagem dos tipos e membros>;
}
struct <identificador> <variável>;
```

No exemplo a seguir, foi criado o registro "ficha_de_aluno" que possui três campos: nome, disciplina e media.

```
struct ficha_de_aluno
{
    char nome[40];
    char disciplina[20];
    float media;
} ;
```

Depois de declarar o registro, precisa-se criar uma variável que vai utilizá-lo. No exemplo, é criada a variável "aluno", que é do tipo "ficha_de_aluno".

```
struct ficha_de_aluno
{
  char nome[50];
  char disciplina[30];
  float media;
};
struct ficha_de_aluno aluno;
```

Para se referir a um campo de um registro, deve-se escrever o nome do registro e o nome do campo separado por um ponto. Nos exemplos a seguir, os campos do registro são acessados e inicializados com valores fixos.

aluno.nome = "Gideon";

aluno.disciplina = "Programação";

aluno.media = 9.5;

A seguir, será apresentado um programa que faz a utilização de registros.

```
#include <iostream.h>

struct ficha_de_aluno //Criação do registro com 3 campos
{
    char nome[50];
    char disciplina[30];
    float media;
 }
aluno;  //Criação da variável aluno que será do tipo struct ficha_de_aluno

int main( )
{
   printf("\nDigite o nome do aluno: ");
   gets(aluno.nome);
   printf("\nDigite o nome da disciplina: ");
   gets(aluno.disciplina);
   printf("\nDigite a media: ");
   scanf("%f", &aluno.media);
   printf("\n\n **** Dados da Struct ****\n\n"); //Impressão dos campos da struct
   printf("Nome do aluno: %s", aluno.nome);
   printf("\nDisciplina .....: %s", aluno.disciplina);
   printf("\nMedia: %f", aluno.media);
}
```

2.3.2 Comando Typedef

Os registros podem ser tratados como um novo tipo de dados (TENEMBAUM et al., 2004, p. 17). Para isso é utilizado o comando typedef. Por exemplo:

typedef struct ficha_de_aluno **aluno**;

aluno a, b;

Depois dessa definição, pode-se passar a dizer "aluno" ao invés de "struct ficha_de_aluno".

2.3.3 Registros e Ponteiros

Cada registro tem um endereço de memória e pode-se imaginar que esse endereço é o de seu primeiro campo. Comumente, é utilizado um ponteiro para armazená-lo e, nesse caso, esse ponteiro aponta para o registro. Por exemplo:

```
aluno *pt;// pt é um ponteiro para o registro ficha_de_aluno
aluno  a;
pt = &a;       // agora pt aponta para a
(*pt).media = 9.3; // tem o mesmo resultado que a.media = 9.3
```

A expressão: *pt->media* é equivalente a (*pt).media = 9.3, sendo muito mais utilizada.

2.4 Recursividade

A recursividade é uma ferramenta de programação muito poderosa, sendo um recurso bastante empregado em linguagens de programação para a solução de problemas computacionais complicados ((TENENBAUM; LANGSAM; AUGENSTEIN, e está diretamente relacionada ao conceito de função e à sua implementação. Essa ferramenta pode ser usada sempre que for possível expressar a solução de um problema em função do próprio problema. Para a implementação de programas recursivos usa-se um procedimento que permite dar um nome a um comando, o qual pode chamar a si próprio.

Um exemplo muito utilizado para a explicação de recursividade é a definição matemática de uma função fatorial, simbolizada pelo sinal de exclamação (!). O fatorial de um número inteiro positivo n é definido como o produto de todos os inteiros entre esse número n e 1. Por exemplo, o fatorial de 6 é igual a: 6 * 5 * 4 * 3* 2 * 1 = 720. O fatorial de 0 e 1 tem valor igual a 1.

Algumas regras matemáticas:

- n! = 1 se n = 0
- n! = n * (n-1) * (n-2) * ... * 1, se n > 0

Assim, o cálculo do valor do fatorial do número 6 pode ser realizado da seguinte forma:

```
Fatorial(6) = 6 *  Fatorial(5)
Fatorial(5) = 5 *  Fatorial(4)
Fatorial(4) = 4 *  Fatorial(3)
Fatorial(3) = 3 *  Fatorial(2)
Fatorial(2) = 2 *  Fatorial(1)
Fatorial(1) = 1
```

Note que o fatorial do número 6 foi obtido através do cálculo do fatorial do número 5, que foi obtido através do cálculo do fatorial de 4, e assim por diante.

Para evitar qualquer abreviatura e um conjunto infinito de definições, pode-se apresentar um algoritmo que aceite um inteiro n e retorne o valor de seu fatorial (TENEMBAUM et al., 2004, p. 133).

O trecho de programa a seguir é chamado iterativo, pois requer a repetição explícita de um processo até que determinada condição seja satisfeita.

```
int fatorial(int n) //Função recursiva que calcula o fatorial
{
  int fati;
  if (n<=1) { //Caso base: fatorial de n <= 1 retorna 1
         return 1;
  else {
        fati = n * fatorial(n-1); //Chamada recursiva
        return (fati);
   }
 }

int main()
{
   int numero,f;
   printf("Digite um numero: ");
   scanf("%d",&numero);
   f = fatorial(numero);   //chamada da função fatorial
   printf("Fatorial = %d",f);
   system("PAUSE");
}
```

Nota-se que o método interativo é mais simples e rápido. A recursividade pode ser usada para a resolução do problema do cálculo do fatorial de um número, assim como pode ser empregada para a resolução de outros problemas, principalmente problemas matemáticos.

Para saber mais

Muitos problemas têm a seguinte propriedade: cada instância do problema contém uma instância menor do mesmo problema. Dizemos que esses problemas têm estrutura recursiva, e no link a seguir você pode acessar a um ótimo conteúdo sobre recursividade.

Disponível em: <http://www.ime.usp.br/~pf/algoritmos/aulas/recu.html>. Acesso em: 12 ago. 2017.

E, assim como no link anterior, o vídeo a seguir explica detalhadamente o assunto; exibe vários exemplos, faz a comparação de uma função não recursiva com uma função recursiva e demonstra o comportamento da função recursiva utilizando a pilha de execução.

Disponível em: <https://www.youtube.com/watch?v=Vg4NhWTCWsI>. Acesso em: 12 ago. 2017.

Nesta seção, você aprendeu sobre ponteiros, que possuem como conteúdo o endereço de memória de outra variável; foi apresentado(a) à alocação dinâmica de memória, através de demonstrações de funções para a alocação e liberação de memória de um computador. Você também aprendeu sobre os registros, que possuem a capacidade de armazenar coleções de dados de diferentes tipos e, também, sobre o conceito de recursividade como uma ferramenta de programação muito poderosa e empregada para a resolução de problemas computacionais complexos.

Atividades de aprendizagem

1. Considerando os conceitos sobre ponteiros e operadores unários, analise o trecho de código a seguir:

```
int a;
a = 10;
int& b = a;
printf("\n%d", a);
printf("\n%d", b);
printf("\n%d", &a);
printf("\n%d", &b);
```

Qual será a sequência de valores que serão impressos na tela?

Obs.: Considere que o endereço de memória da variável "a" é **AE14Z**.

a) 10, 10, AE14Z, AE14Z.
b) AE14Z, AE14Z, 10, 10.
c) 10, AE14Z, 10, AE14Z.
d) AE14Z, 10, AE14Z, AE14Z.
e) 10, 10, 10, 10.

2. Os registros são utilizados quando há a necessidade de armazenamento de informações que possuem dados de diferentes tipos. Implemente um programa C que utilize um registro para armazenar as seguintes informações de um livro: código e quantidade de páginas. Você pode inicializar os campos desse registro no programa principal. Para exibir os seus dados, crie uma função chamada "exibir".

Fique ligado

Nesta unidade, você começou seu estudo aprendendo sobre os principais conceitos relacionados às Estruturas de Dados por meio de analogias com o seu cotidiano; em seguida, foi apresentado(a) à definição de funções e soube como elas podem ajudar para que o código de um programa fique mais compreensível e organizado. Você aprendeu que uma função pode retornar apenas um valor e que se houver a necessidade de que os valores das variáveis sejam alterados no programa principal, deve passar esses valores por referência; soube o que são ponteiros e como eles são fundamentais para a implementação de estruturas de dados dinâmicas, já que armazenam o endereço de memória de outras variáveis, e, por fim, soube o conceito e a importância de recursividade e como essa ferramenta pode ser útil para se implementar funções complexas e que exigem maior carga computacional.

Para concluir o estudo da unidade

Na Computação, a Estrutura de Dados é utilizada para resolver a maioria dos problemas complexos relacionados à programação, e é de extrema importância conhecer e saber implementar as estruturas básicas para armazenamento de dados a fim de uma maior eficiência e rapidez na execução dos programas.

Saber qual o tipo de estrutura de dados que deve ser implementado para a resolução de um problema específico é de extrema importância, mas, para isso, você deve primeiramente entender suas características e adquirir habilidades para a sua implementação em uma linguagem de programação. Nesta unidade, algumas técnicas de programação em C foram apresentadas e é fundamental que você pratique os conceitos estudados, principalmente sobre ponteiros e funções. Tente exercitar e replicar os exemplos que foram estudados, assim como desenvolver novos programas utilizando essas técnicas de programação em C.

Atividades de aprendizagem da unidade

1. Para se programar em qualquer linguagem são utilizadas variáveis para o armazenamento de dados. Cada variável possui um tipo de dado específico, dependendo de sua finalidade; e em relação a tais tipos, analise as seguintes afirmativas:
(I) O tipo de dados na perspectiva computacional é entendido como métodos de interpretação da memória do computador, ou seja, o que ele pode fazer.
(II) Se o tipo de dados for dissociado do computador ou da máquina, ele é chamado de Tipo Abstrato de Dados – TAD.
(III) A Estrutura de Dados (ED) é a maneira de se implementar um Tipo Abstrato de Dados (TAD).
Assinale a alternativa correta.
a) Apenas a afirmativa II está correta.
b) Apenas as afirmativas I e II estão corretas.
c) Apenas as afirmativas I e III estão corretas.
d) Apenas as afirmativas II e III estão corretas.
e) As afirmativas I, II e III estão corretas.

2. Uma função é um conjunto de instruções desenhadas para cumprir uma tarefa particular e agrupadas numa unidade com um nome para referenciá-la. Considere o seguinte programa:

```
int Funcao1 ( )
{
    int a, b, x;
    printf (“Digite um numero: \n“);
    scanf(“%d”, &a);
    printf (“Digite um numero: \n“);
    scanf(“%d”, &b);
    x = a * b;
    printf(“\nResultado = %d“, x);
}

int main ()
{
   Funcao1 ( );
   system(“PAUSE”);
 }
```

Em relação ao programa, analise as afirmativas e as classifique como Verdadeiras (V) ou Falsas (F):

I) A função desse programa é invocada no programa principal através do comando "Funcao1 ();".
II) Com a chamada de função, a execução do programa principal para e a função é iniciada, criando duas variáveis do tipo inteiro.
III) Ao final da execução de todos os comandos da função, a execução do programa retorna à linha posterior, à chamada da função no programa principal.
Assinale a alternativa que contém a sequência correta.
a) V-V-V.
b) F-V-V.
c) V-F-V.
d) V-F-F.
e) F-F-F.

3. A passagem por referência utiliza um operador, chamado de operador unário de referência, e é simbolizado por "&". Considere as instruções:
int a;
int & x = a;
Em relação a essas instruções, analise as afirmativas:
I) O operador &x = x indica que agora x é outro nome para a.
II) As operações nas duas variáveis (a e x) não têm o mesmo resultado.
III) O operador "&" faz referência ao conteúdo de uma variável.
IV) Uma referência não é uma cópia da variável a qual se refere, mas sim, a mesma variável sob diferentes nomes.
Assinale a alternativa correta:
a) Apenas a afirmativa I está correta.
b) Apenas as afirmativas I e IV estão corretas.
c) Apenas as afirmativas I e III estão corretas.
d) Apenas as afirmativas II e III estão corretas.
e) As afirmativas I, II, III e IV estão corretas.

4. Nesta unidade, foi estudado o conceito de registro, que armazena vários campos com tipos de dados diferentes. Considere o seguinte programa:

```
#include <iostream.h>

typedef struct {
    int matricula;
    char nome[100];
    float nota1;
    float nota2;
} Aluno;
```

```
int main()
{
	Aluno alunos[3];
	printf("Dados: nome, matricula, nota1, nota2\n");
	for(int i=0; i < 3; i++){
		printf("\nInforme os dados do aluno(%i): ",i+1);
		scanf("%s %i %f %f",alunos[i].nome, &alunos[i].matricula,
			&alunos[i].nota1, &alunos[i].nota2);
	}

	printf("\nMatricula\tNome\tMedia\n");
	for(int i=0; i < 3; i++) {
		 printf("% f\n",alunos[i].matricula,alunos[i].nome,(alunos[i]. nota1 +
alunos[i]. nota2) / 2);
	}

	system(“PAUSE”);
}
```

Considerando o código apresentado, qual a finalidade desse programa?

a) Encontrar a menor média de um aluno.
b) Encontrar a maior média de um aluno.
c) Calcular e exibir as médias de três alunos.
d) Encontrar os valores pares das matrículas dos alunos.
e) Encontrar os valores ímpares das matrículas dos alunos.

Referências

JUNIOR, Dilermando Piva; et al. **Estrutura de Dados e Técnicas de Programação**. Rio de Janeiro: Elsevier-Campus, 2014.

MIZRAHI, Viviane Victorine. **Treinamento em linguagem C++**. São Paulo: Makron, 2006.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: Pearson Makron Books, 2004.

VELOSO, Paulo; et al. **Estrutura de dados**. Rio de Janeiro: Campus, 1986.

Tipos e estruturas de dados

Nathalia dos Santos Silva

Objetivos de aprendizagem

Nesta unidade, você será apresentado aos tipos de dados pertinentes ao desenvolvimento de sistemas e a suas características principais. O objetivo é que você conheça esse tema com detalhes e, a partir disso, possa tomar boas decisões em seus projetos, no que tange à escolha e definição da estrutura de dados utilizada.

Você também conhecerá os tipos de dados existentes e aqueles possíveis de serem implementados na linguagem C - uma linguagem popularmente conhecida e que serve de base para outras - bem como os comandos, as funções e a sintaxe dessas implementações.

Por fim, ao conhecer os tipos de dados abstratos e compostos, homogêneos e heterogêneos, você será capaz de criar estruturas adequadas para seus programas utilizando dados primitivos.

Seção 1 | Vetores

Na Seção 1, estudaremos os vetores - uma estrutura de dados vastamente utilizada e disponível em praticamente todas as linguagens de programação. Sua ampla utilização se deve às vantagens de representação, que também serão abordadas, e ao fato de ser uma estrutura aplicável em diversos tipos de dados primitivos.

Seção 2 | Matrizes

Na Seção 2, estarão presentes os conceitos envolvendo matrizes (que são extensões dos vetores e muito úteis em aplicações comerciais), uma vez que são utilizadas em agrupamento de dados com mais de uma referência de classificação, e em aplicações de processamento de imagens ou outras estruturas multidimensionais.

Seção 3 | Tipos de Dados

Na Seção 3, veremos tipos de dados adicionais criados para atenderem uma demanda de maior organização na manipulação de dados, classificados conforme o nível de detalhes e especificações que carregam, passando por dados abstratos, compostos e heterogêneos.

Introdução à unidade

Esta unidade traz os conceitos sobre os principais tipos avançados de dados e de como manipulá-los, bem como sua sintaxe, por meio da linguagem de programação C. Vamos nos aprofundar nos tipos de dados avançados, além daqueles dados primitivos que você já estudou, com o objetivo de abranger uma classe maior de representação para serem utilizados em problemas reais.

Desde o surgimento dos computadores e sistemas de informação, os tipos de dados existentes têm evoluído, a grande maioria deles são muito funcionais e, por isso, estão descritos nesta unidade.

As operações que realizamos por meio dos comandos de uma linguagem de programação estão diretamente relacionadas às operações suportadas pelos dados que escolhemos, por isso, é importante que conheçamos as possibilidades e limitações dos tipos de dados (EDELWEISS; GALANTE, 2009).

Como os dados são a base dos sistemas de programação, eles devem estar inseridos em nossas escolhas quanto à definição dos aspectos de implementação, até mesmo por influenciarem na simplicidade, complexidade e na organização do código.

Algumas representações são óbvias e não causam dúvidas, como o tipo int escolhido para um contador, o tipo float para altura de uma pessoa, entre outros frequentemente utilizados; entretanto, algumas informações podem necessitar de uma pequena avaliação para serem bem representadas, por exemplo: a data de nascimento é um dado numérico? É interessante que eu a imprima no formato --/--/--? Ela deve ser decomposta em dia, mês e ano para algum cálculo de verificação de idade?

As respostas para essas perguntas podem variar conforme a necessidade do projeto e influenciar na escolha do tipo de dados. Por exemplo, se a data for utilizada para algum cálculo, é interessante que ela seja numérica, se for só um dado informativo, pode ser um *string*, e ela pode, inclusive, ser armazenada no formato para impressão. Seguindo o mesmo exemplo, caso fosse necessário, fazer um cálculo para determinar a idade de uma pessoa, ou ainda caso fosse interessante apresentar a data em formatos diferentes, o ideal seria armazenar os

valores da data em três variáveis, e não somente uma contendo tudo.

Visto isso, a decisão sobre a escolha dos dados deve ser tomada conforme a necessidade levantada, e ainda, deve buscar a solução menos custosa quanto à utilização de memória, ao mesmo tempo em que garanta produtividade em seu desenvolvimento (SZWARCFITER; MARKENZON, 2015)

Seção 1

Vetores

Introdução à seção

Você já observou como as construções verticais, como edifícios, permitem agrupar um número maior de moradores em um mesmo endereço (rua e número)? A estrutura de vetores é muito semelhante a essa forma de alocação! Com um mesmo identificador, ou seja, o nome de variável, é possível referenciar um maior conjunto de dados, desde que sejam de mesmo tipo. E como os apartamentos, os vetores possuem uma organização interna.

Os vetores são estruturas de dados comumente utilizados para melhor organizar variáveis relacionadas entre si conceitualmente; por serem versáteis, podem admitir diversos tipos primitivos. Esta seção abordará como são tratados os vetores na linguagem C, sua sintaxe, os principais comandos e as operações permitidas sobre esses dados.

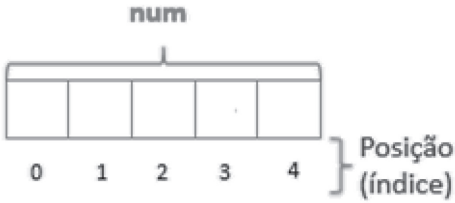
1.1 Sintaxe para declaração de Vetores

Os vetores permitem que dados de mesmo tipo e mesmo significado conceitual sejam agregados por um mesmo nome de variável (PEREIRA, 2016), como o exemplo citado dos edifícios. Internamente, o responsável pela organização e identificação de cada item é o índice, que deve sempre ser um número inteiro, ou seja, o int. No entanto, quanto ao tipo de dados do vetor, esse pode ser *int, float, string, double, bool ou char*; ou seja, os tipos de dados primitivos básicos aceitos pela linguagem C.

Os vetores também são conhecidos como estruturas de dados estáticas, pois seu tamanho permanece o mesmo em tempo de execução (TENENBAUM; LANGSAM; MOSHE, 1995).

Imagine: se pudéssemos descrever fisicamente um vetor, ele teria a forma ilustrada na Figura 2.1:

Figura 2.1 | Vetor "num"



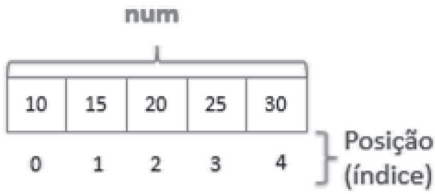
Fonte: elaborada pela autora.

Para obtermos o vetor da Figura 1.1, iniciaremos pela declaração do vetor:

```
int num[5];
```

Neste exemplo foi criado um vetor de 5 posições para inteiros, chamado num. Podemos afirmar que esse vetor é do tipo int. e iniciar com um vetor já preenchido, como na Figura 2.2:

Figura 2.2 | Vetor "num" preenchido



Fonte: elaborada pelo autor.

Para isso, devemos implementar o seguinte código:

```
int num[5] = {10, 15, 20, 25, 30};
```

Um vetor pode ter qualquer valor inteiro em cada posição alocada. Para demonstrar a alocação de vetores de outros tamanhos e tipos, mantemos a mesma sintaxe com os respectivos parâmetros:

```
float valor [10];
char resposta [3];
```

Esse trecho implementa a declaração de 2 vetores, sendo a primeira linha o vetor de 10 elementos em ponto flutuante com identificado valor; o segundo, um vetor resposta para armazenar 3 caracteres.

Da mesma forma que variáveis podem ser declaradas e atribuídas em um mesmo comando, vetores também:

```
float valor [10] = {3, 4};
char resposta [3] = {'a', 'c', 'd', 'a', 'b'};
```

Observe que podemos tentar atribuir mais ou menos elementos que a quantidade declarada no vetor, e isso não foi impedido pelo compilador, o que é uma desvantagem, pois pode causar erros de lógica ou acessar espaços indevidos de memória, e tudo ser imperceptível ao programador.

Um vetor também pode ter seu tamanho declarado através de seus elementos:

```
int n[ ] = {1, 2, 4, 8, 10, 12};
```

Outra desvantagem desse tipo de declaração é não ser explícita a quantidade de elementos do vetor.

1.2 Sintaxe para acessar elementos do vetor

É importante ressaltar que um vetor de tamanho **n** possui **posições** determinadas pelos **índices** de 0 ao n-1; ou seja, um vetor de 5 posições manipula do índice 0 ao 4; o índice, então, é o valor que indica a posição que queremos acessar no vetor:

```
int nun[5] = {10, 15, 20, 25, 30};

X = num [3];
```

Nesse comando, estamos atribuindo à variável x o conteúdo do vetor na posição 3, ou seja, a 4ª posição da esquerda para direita, que é igual ao valor 25.

Podemos, também, inserir um valor em um elemento específico do vetor, aliás, é a única maneira de alterarmos o valor do vetor ao longo do código, depois de sua declaração. A sintaxe é análoga à atribuição de dados simples:

```
nun[2] = 17;
```

Essa sintaxe representa a ação de inserir o valor 17 na posição 2 do vetor.

Para saber mais

Os valores contidos nos vetores permitem as mesmas operações que seus tipos, e como são versáteis, permitem várias possibilidades a partir da lógica com seus índices e valores. Você pode treinar mais sobre as operações em vetores: <https://www.ime.usp.br/~macmulti/exercicios/vetores>. Acesso em: 10 ago. 2017.

Questão para reflexão

Nos exemplos apresentados foram criados vetores de inteiros e ponto flutuante. Podemos criar vetores de *strings* em linguagem C? O conceito de vetor será mantido?

1.3 Sintaxe com laços para percorrer o vetor

Como podemos perceber, em todas as operações que fazemos com vetores em C é feito elemento a elemento, então, sempre utilizaremos um laço para percorrer os vetores através dos elementos, um a um, por meio de um índice inteiro e sempre representado por i.

Também, é através do laço que imprimimos os valores dos elementos do vetor na tela, um a um, controlado pelo seu índice.

Pensando em uma boa prática de programação, o ideal é utilizar o

comando define, dada a necessidade de se alterar o tamanho do vetor e de todas as funções a ele pertinentes, alteramos este valor somente uma vez no início do código, como no código da Figura 2.3, em que o controle do laço segue automaticamente o valor do comando define:

Figura 2.3 | Código de inserção e impressão de valores nos vetores

```
#include <stdlib.h>
#define TAMANHO 10
int main()
{

    int inteiros[TAMANHO];

    for(i = 0; i < TAMANHO; i++)
    {
        inteiros[i] = i;
    }

    for(i = 0; i < TAMANHO; i++)
    {
    printf("  %d  ", inteiros[i]);
    }
    return 0;
}
```

Fonte: elaborada pela autora.

Como explicado, o processo de preenchimento do vetor, seus cálculos e a impressão na tela são feitos elemento a elemento, normalmente por meio de um laço for.

A seguir, traremos um código (na Figura 2.4) que preenche diferentes vetores, para que você conheça algumas possibilidades e, posteriormente, possa adaptá-las conforme sua necessidade.

Figura 2.4 | Código de inserção e impressão de valores nos vetores

```c
#include <stdio.h>
#include <stdlib.h>
#define TAMANHO 10

int main()
{
    int inteiros[TAMANHO];
    int zeros[TAMANHO];
    int quadrado[TAMANHO];
    int dobro[TAMANHO];
    int i;

    for(i = 0; i < TAMANHO; i++)
    {
        inteiros[i] = i;
        zeros[i] = 0;
        quadrado[i] = i*i;
        dobro[i] = i * 2;
    }

    printf(" Vetores: \n ");
    printf(" Indice - Zeros - Inteiros - Quadrado - Dobro\n ");

    for(i = 0; i < TAMANHO; i++)
    {
        printf("%d  -- %d  -- %d  -- %d  -- %d ",
               i, zeros[i], inteiros[i], quadrado[i], dobro[i]);
    }

    return 0;
}
```

Fonte: elaborada pela autora.

Exemplo 2.1:

Considere uma loja que paga uma comissão de 5% aos seus funcionários e precisa calcular a comissão de um ano, conforme as vendas de cada um. Implemente um programa que atenda a essa necessidade.

Como é um problema de diversos dados (comissão ao longo dos 12 meses) e possuem certa semelhança e significado, podemos e devemos utilizar vetores.

O código apresentado na Figura 2.5 apresenta a solução em C para esse problema.

Figura 2.5 | Solução do Exemplo 1.1

```
#include <stdio.h>
#include <stdlib.h>
#define MESES 4
#define FILIAIS 3

int main()
{
    float vendas[MESES][FILIAIS];
    float receber[MESES][FILIAIS];
    float comissao = 0.05f;
    int i,j;

    for(i = 0; i < MESES; i++)
        for(j = 0; j < FILIAIS; j++)
        {
            printf("\nDigite as vendas do %do mes, da filial %d: ", i+1, j+1);
            scanf("%f", &vendas[i][j]);
            receber[i][j] = vendas[i][j] * comissao;
        }

    printf("\nOs valores a receber sao: ");

    for(j = 0; j < FILIAIS; j++)
    {
        printf("\nNa filial %d:", j+1);
        for(i = 0; i < MESES; i++)
        {
            printf("\n %.2f no %do mes;", receber[i][j], i+1);
        }
    }

    return 0;
}
```

Fonte: elaborada pela autora.

Atividades de aprendizagem

1. Ao trabalharmos com vetores, precisamos saber controlar seus índices - esse é um passo essencial. Sabendo disso, analise as afirmações a seguir:

I – A primeira posição de um vetor é sempre identificada por [0].

II – Um vetor de 5 posições permite o acesso à sua última posição através do índice [5].

III – A soma dos elementos de dois vetores, A e B, é realizada pelo código A[ ] + B[ ].

Assinale a alternativa correta:

a) Somente I está correta.

b) Somente II está correta.

c) Somente III está correta.

d) Somente I e II estão corretas.

e) Somente I e III estão corretas.

2. Bruno precisa de um programa que trabalhe com os dados de sua pesquisa; precisa armazenar informações sobre 100 amostras e, posteriormente, fazer uma média aritmética delas. Assinale a alternativa que contém a declaração adequada e correta das estruturas de dados que ele usará:

a) int amostras[100] ; int media.
b) int amostras; char media[100].
c) float amostras[100] ; float media.
d) float amostras[100] ; int media.
e) char amostras[100] ; float media[100].

Seção 2

Matrizes

Introdução à seção

Você percebeu que vetores são muito úteis para a programação de algoritmos ao agruparem diversos dados em uma estrutura de variável com o mesmo nome. Agora, saiba que podemos estender esse comportamento e criar matrizes! São semelhantes aos vetores em algumas características básicas e possuem uma notação que as identifica como matrizes.

2.1 Sintaxe para declaração de matrizes

As matrizes podem ser abstraídas como vetores bidimensionais, ou seja, um conjunto de dados de mesmo tipo organizados de maneira estruturada e reconhecidos por um mesmo identificador. Como estamos criando uma estrutura de dados de duas dimensões, com **linha** e **coluna**, precisamos de dois argumentos para dimensionar a matriz e dois índices de controle.

Colocando um adendo no parágrafo anterior, a matriz pode ser uma abstração de duas dimensões para facilitar o entendimento dos programadores, porque, em C, a matriz é armazenada e acessada de maneira linear (MIZRAHI, 2008).

A sintaxe para declaração da matriz é:

```
int matriz [2] [3];
```

2.2 Sintaxe para manipulação de matrizes

Como criamos laços para acessar os elementos dos vetores, um a um, com as matrizes é análogo, assim como as atribuições.

Vamos analisar, na prática, como ficaria o nosso Exemplo 1.1 de vendas e comissão, caso tivéssemos mais de um vendedor, através da Figura 2.6:

Figura 2.6 | Código de matrizes

```
#include <stdio.h>
#include <stdlib.h>
#define MESES 12

int main()
{
    float vendas[MESES];
    float receber[MESES];
    float comissao = 0.05f;
    int i;

    for(i = 0; i < MESES; i++)
    {
        printf("\nDigite as vendas do %do mes: ", i+1);
        scanf("%f", &vendas[i]);
        receber[i] = vendas[i] * comissao;
    }

    printf("\nOs valores a receber sao: ");

    for(i = 0; i < MESES; i++)
    {
        printf("\n %.2f no %do mes;", receber[i], i+1);
    }

    return 0;
}

```

Fonte: elaborada pela autora.

Como toda atividade de implementação, é sugerido que você compile o programa e, então, faça alterações na lógica conforme a necessidade!

Para saber mais

Alguns programas, como o Scilab ou o Matlab, são desenvolvidos para trabalharem especialmente com matrizes, dada a versatilidade e aplicabilidade desse conceito tão utilizado na programação! Você pode conhecer mais sobre isso em: <http://www.mat.ufmg.br/~espec/tutoriais/scilab>. Acesso em: 10 ago. 2017.

Questão para reflexão

Podemos ter matrizes com mais de 2 dimensões?

Atividades de aprendizagem

Para responder às questões 1 e 2, considere o código a seguir:

```
#include <stdlib.h>
#define TAMANHO 10
int main()
{
    int inteiros[TAMANHO];
    for(i = 0; i < TAMANHO; i++)
    {
        inteiros[i] = i;
    }
    for(i = 0; i < TAMANHO; i++)
    {
    printf("  %d  ", inteiros[i]);
    }
    return 0;
}
```

1. O mais importante na manipulação de matrizes é controlar seus laços e índices, sabendo disso, e com base no código apresentado, assinale a alternativa correta:

a) A matriz possui 6 elementos.
b) A matriz possui 20 elementos.
c) O valor impresso será 2.
d) O valor impresso será 6.
e) Será impressa a matriz toda.

2. Ainda com base no código apresentado, assinale a alternativa incorreta:

a) O elemento matriz[0][0] não teve atribuição de valores.
b) O comando matriz[4][4] = 10 apesar de inválido pode ser aceito pelo compilador.
c) O valor 9 não aparece na matriz em nenhum elemento.
d) O elemento matriz[3][4] é igual a 12.
e) Após o comando x = matriz[2][2], x passa a valer 4.

Seção 3

Tipos de Dados

Introdução à seção

Um tipo de dado é um conceito que reúne informações sobre um conjunto que pode ser representado com aquelas especificações, por exemplo, o tipo char reúne especificações de caracteres.

Como boa parte dos conceitos computacionais, eles estão diretamente envolvidos com o hardware do computador e com propriedades lógicas sobre ele, ou seja, as operações e cálculos suportados.

O conceito que discutiremos nesta seção é uma especificação de como um novo tipo é representado por objetos de tipos de dados já existentes e de como será manipulado pelo software (TENENBAUM; LANGSAM; MOSHE, 1995).

Esta seção apresenta os tipos de dados que podemos encontrar para implementar soluções através de algoritmos em linguagem C e, portanto, a dividimos em Dados do tipo: Abstrato, Complexo e Heterogêneo; e vamos discutir cada um deles.

3.1 Tipos de Dados Abstratos

Do inglês *Abstract Data Types* (ADT), ou sua sigla em português TDA, é uma **representação de dados** acompanhada das **operações** que se pode fazer com eles (DEITEL; DEITEL, 2011). As variáveis passam a ser mais do que acessórios na programação, elas descrevem o cenário a ser modelado, isso é, estão mais próximas em descrever o mundo real e menos voltadas aos detalhes específicos de implementação.

O conceito de Abstração de Dados é muito semelhante a uma caixa-preta com entrada e saída de informações, justamente por representar de maneira fiel as necessidades levantadas e o que precisa ser realizado, sem detalhar como o compilador trata as particularidades dos dados em si.

Deitel e Deitel (2011) exemplificam que o computador não tem uma referência de significado para os tipos matemáticos como int, float ou double, por exemplo; ele apenas possui mecanismos para representar e processar informações pertinentes a tais dados de uma

maneira que sejam viáveis e fisicamente capazes de relacionar com as características do que conhecemos que seja um número inteiro ou um ponto flutuante, de precisão simples ou dupla, respectivamente.

Sempre que esse tipo é escolhido em nossa implementação, devemos conhecer as operações que podemos realizar sobre ele e se possuem alguma restrição ou limitação, já que nem todos os tipos matemáticos podem ser implementados em todas as máquinas a partir dos programas (TENENBAUM; LANGSAM; MOSHE, 1995). Então, podemos entender que os tipos de dados abstratos são maneiras de representar os conjuntos que conhecemos no mundo real em um nível de abstração que seja adequado para o computador.

Adicionalmente, utilizamos tipos de dados abstratos em estruturas de dados como vetores, filas e pilhas, detalhadas ao longo deste livro, com os quais não precisamos nos preocupar devido à forma como esses dados são operados pelo hardware, só precisamos saber utilizar as operações, como inserir ou retirar, sobre os elementos.

Em implementações com C, você pode criar tipos abstratos com *structs* e os tipos *typedef*; já em C++, C# ou Java você também pode criar seus próprios tipos abstratos através de classes.

3.2 Tipos Compostos de Dados

Os tipos compostos de dados são formados a partir dos tipos primitivos de dados, ou seja, daquelas estruturas mais simples da linguagem, como os tipos *int, float, double, ou char*, organizados e combinados de maneira a formarem uma nova estrutura, também são conhecidos como tipos derivados (PINHEIRO, 2012).

Dois desses tipos são o vetor e a matriz, que vimos anteriormente. Eles são uma estrutura de dados homogêneos, ou seja, todos seus integrantes são do mesmo tipo, e possuem uma capacidade limitada de elementos; no entanto, por limitada não entendamos como pouca ou pequena, e sim como um valor conhecido.

String é um outro tipo de dado composto, pois trata-se de um conjunto ordenado de caracteres e, por ser um vetor em C, é tratada como ponteiro.

As strings podem armazenar uma série de caracteres entre letras, números, caracteres especiais, sequência de caracteres com espaços, formando uma frase etc. Por ser tratada com um dado somente, daí sua definição de dado composto.

O tamanho da string pode ser de qualquer comprimento; ela deve,

sempre, terminar com o caracter nulo ('\0', leia-se barra invertida zero) e se tentarmos armazenar na string um valor de caracteres maior àquele declarado, os dados sobressalentes irão sobrescrever outros dados em posições da memória que sucedem o local de memória alocado para essa string (DEITEL; DEITEL, 2011).

Podemos definir a string da seguinte maneira:

```
char nome[] = "Nathalia";
char nomeItem[] = {'N', 'a', 't', 'h', '\0'};

printf("%s\n", nome);
printf("%s\n", nomeItem);
```

A saída resultante será:

Nathalia

Nath

Se esquecermos o '\0' ao final dessa declaração de chars (na segunda declaração), a variável conterá dados de memória não desejáveis, o que também, usualmente, chamamos de lixo de memória.

Quanto à entrada de dados, o comando scanf não lê espaços ou mudanças de linha e é limitado em 19 caracteres. Ou seja, para um sobrenome com dois nomes, não seria possível adotá-lo:

```
char nome[] = "Nathalia";
char sobrenome[30];

scanf("%s", sobrenome);

printf("%s\n", nome);
printf("%s\n", sobrenome);
```

Para uma entrada no scanf no formato "da Silva", o tipo string só armazenaria o "da", sendo assim, precisamos de funções específicas que tratem strings, como **getchar**, ou outras contidas na biblioteca stdio.h (MIZRAHI, 2008):

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char lido;
    char frase[50];
    int i=0;

    printf("Digite a frase:");

    while ( (lido = getchar()) != '\n' )
    {
        frase[i++] = lido;
    }
    frase[i] = '\0';

    printf("A frase digitada foi:");
    printf("%s", frase);

    return 0;
}

```

Para saber mais

O conteúdo de strings é muito amplo e os comandos para manipulá-las são muitos, principalmente devido à flexibilidade de se trabalhar com um vetor; aliás, poderíamos ter um outro capítulo escrito somente para lidar com as strings. No entanto, você pode conhecer mais sobre o assunto no livro *C: como programar,* de Paul Deitel e Harvey Deitel.

Um outro tipo composto de dados, que é utilizado na linguagem C, é a **estrutura**, ou struct, também conhecida como registro; ela possui um conjunto de membros, sendo cada membro um dado com identificador próprio, também chamado de campo.

Pode ser declarado da seguinte maneira (TENENBAUM; LANGSAM; MOSHE, 1995):

```c
struct{
char nome[15];
char sobrenome[30];
} autor1, autor2;
```

Sendo nome e sobrenome os membros da estrutura, e autor1 e autor2 as variáveis criadas desse tipo. Como uma forma mais completa e intuitiva, incentivando a melhor organização do código, também podemos criar um nome para a estrutura, e assim declarar as estruturas na linha seguinte:

```
struct nome_completo{
char nome[15];
char sobrenome[30];
} autor1, autor2;

struct nome_completo autor1;
```

Observe que as duas declarações resultam no mesmo objetivo.

Ainda é possível que criemos um tipo composto abstrato de dados, isso é, apliquemos os conceitos de TDA para obter esta estrutura:

```
typedef struct {
char nome[15];
char sobrenome[30];
} TDA_NOME_COMPLETO;

TDA_NOME_COMPLETO autor1, autor 2;
```

Seus membros são acessados da seguinte maneira:

```
autor1.nome = "Gisele";
```

Criamos um programa que imprime o nome completo e o nome abreviado das pessoas - como utilizamos em referências bibliográficas para autores de livros. Acompanhe e, se possível, implemente o código descrito na Figura 2.7:

Figura 2.7 | Programa em C para operar structs

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

int main()
{
   struct nome_completo{
   char nome[15];
   char sobrenome[30];};

   struct nome_completo autor1, autor2;

   char lido, primeiraletra;
   int i = 0; //contador para auxiliar

   printf("Digite o primeiro nome do autor: \n");
   while ( ( lido = getchar() ) != '\n' )
        autor1.nome[i++] = lido;
    autor1.nome[i] = '\0';
    primeiraletra = autor1.nome[0];

    printf("Digite o sobrenome do autor: \n");
    i = 0;
    while ( ( lido = getchar() ) != '\n' )
        autor1.sobrenome[i++] = lido;
    autor1.sobrenome[i] = '\0';

    printf("Nome completo do autor: \n");
    printf("%s %s", autor1.nome, autor1.sobrenome);
    printf("\nNome abreviado do autor: \n");
    printf("%s, %c.", autor1.sobrenome, primeiraletra);

    return 0;

}

```

Fonte: elaborada pela autora.

Questão para reflexão

É possível e viável criar um vetor de structs?

3.3 Tipos de dados Heterogêneos

Da mesma forma que criamos *structs* com um único tipo - no caso, o tipo *char* -, poderíamos ter criado *structs* com o único tipo *int* ou *float*, e seriam todas consideradas structs homogêneas, ou seja, um tipo de dados homogêneos. Ao mesmo tempo, se quiséssemos criar uma struct que contivesse tipos distintos de dados, como os tipos int

e char, essa estrutura de dados passaria a ser uma estrutura de dados heterogêneos, ou uma coleção heterogênea de dados (PEREIRA, 2016). Esse tipo de dados também é conhecido na linguagem C como registro, e pode ser implementado como a definição a seguir, na Figura 2.8:

Figura 2.8 | Struct de dados heterogêneos

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

int main()
{
   struct nome_completo{
   char nome[15];
   char sobrenome[30];};

   struct nome_completo autor1, autor2;

   char lido, primeiraletra;
   int i = 0; //contador para auxiliar

   printf("Digite o primeiro nome do autor: \n");
   while ( ( lido = getchar() ) != '\n' )
        autor1.nome[i++] = lido;
    autor1.nome[i] = '\0';
    primeiraletra = autor1.nome[0];

    printf("Digite o sobrenome do autor: \n");
    i = 0;
    while ( ( lido = getchar() ) != '\n' )
        autor1.sobrenome[i++] = lido;
    autor1.sobrenome[i] = '\0';

    printf("Nome completo do autor: \n");
    printf("%s %s", autor1.nome, autor1.sobrenome);
    printf("\nNome abreviado do autor: \n");
    printf("%s, %c.", autor1.sobrenome, primeiraletra);

    return 0;

}

```

Fonte: elaborada pela autora.

Repare que a grande vantagem desse tipo de dados é a organização que ele possibilita ao programador: os dados dos registros possuem o mesmo prefixo. Além do mais, se houvesse a necessidade de cadastrar diversos jogadores, e não fosse usado struct, teríamos uma quantidade muito maior de variáveis para manipular.

Atividades de aprendizagem

1. (Adaptado de ENADE, 2011) O conceito de Tipo de Dados Abstrato (TDA) é popular em linguagens de programação. Nesse contexto, analise as afirmativas a seguir:

I. A especificação de um TDA é composta das operações aplicáveis a ele, da sua representação interna, e das implementações das operações.

II. Se S é um subtipo de outro T, então entidades do tipo S em um programa podem ser substituídas por entidades do tipo T, sem alterar a corretude desse programa.

III. É possível construir TDA a partir de dados compostos.

É correto apenas o que se afirma em:

a) I.
b) II.
c) III.
d) I e II.
e) I e III.

2. Considere como base o programa implementado na Figura 2.7. O trecho do programa que imprime o nome abreviado do autor poderia ter sido implementado de uma outra maneira, ainda equivalente.

```
printf("\nNome abreviado do autor:\n");
printf("%s, %c", autor1.sobrenome, ** );
```

Analise as afirmativas a seguir e assinale aquela que apresenta o conteúdo correto no lugar dos símbolos:

a) autor1.nome[0].
b) nome[0].
c) autor1.primeiraletra.
d) autor1.primeiraletra[0].
e) Autor0.primeiraletra[1].

Fique ligado

Você viu nesta unidade como os dados, em suas diferentes formas, influenciam a programação e que os tipos existentes podem vir a formar novos tipos de dados, sempre que nossa aplicação necessitar. Não deixe de resolver os exercícios e se desafie a implementar os exemplos! É a prática dos conceitos que vai fazê-lo adquirir afinidade com os tipos de dados e entender como utilizá-los.

Para concluir o estudo da unidade

Você deve estudar os conceitos sobre os tipos de dados, afinal, eles explicam como são formadas as diversas possibilidades que encontramos quando nos referimos a dados. Embora a sintaxe e as limitações acerca dos dados possam variar de linguagem para linguagem, conhecer os conceitos em uma como a C permite que você entenda melhor os outros tipos de dados de outras linguagens derivadas, como C++ e C#.

Atividades de aprendizagem da unidade

1. Os elementos de um array são relacionados entre si pelo fato de que possuem o mesmo _____ e _____.
Assinale a alternativa que preenche corretamente as lacunas:
a) Tipo; identificador.
b) Dado; estrutura.
c) Tipo; valor.
d) Valor; identificador.
e) Dado; local na memória.

2. Considere o código a seguir:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       int matriz[10][4];
7       int x = 6;
8
9       matriz[10][4] = 45;
10
11      return 0;
12  }
```

Assinale a alternativa correta:
a) A linha 9 apresenta um erro.
b) O número 10 é o conteúdo da variável matriz na parte alta.
c) X é o índice.
d) 45 é o conteúdo de um elemento da matriz.
e) O elemento matriz[10][4] é igual ao elemento matriz[45].

3. Considere a declaração a seguir:

```
char nome[] = "Nathalia";
char nomeItem[] = {'N', 'a', 't', 'h', '\0'};

printf("%s\n", nome);
printf("%s\n", nomeItem);
```

Assinale a alternativa correta:
a) nomeItem[1] tem valor N.
b) nome[ ] é um dado do tipo primitivo.
c) nomeItem é uma matriz de 5 dimensões.
d) A primeira impressão na tela é somente N.
e) nome é um vetor de caracteres.

4. Analise as informações a seguir e classifique-as como verdadeiro (V) ou falso (F):
( ) Um mesmo vetor armazena valores de tipos distintos.
( ) Uma mesma matriz armazena valores de tipos distintos.

( ) Uma struct organiza valores de tipos distintos.
( ) Uma struct é um tipo de dado primitivo.
( ) Um vetor de inteiros é um tipo composto.
a) F – F – V – F – V.
b) F – V – V – F – V.
c) V – V – F – F – V.
d) V – V – V – F – V.
e) F – F – V – V – V.

5. As operações que realizamos, por meio dos comandos de uma linguagem de programação, estão diretamente relacionadas às operações suportadas pelos dados que escolhemos, por isso, é importante que conheçamos as possibilidades e limitações dos tipos de dados (EDELWEISS; GALANTE, 2009).
Analise as alternativas a seguir e assinale a incorreta:
a) É possível implementar um vetor de um tipo composto de dados.
b) Uma matriz é um elemento linear, porém composto.
c) Para trabalhar com tipo de dados abstrato devemos conhecer sua estrutura interna.
d) Um tipo char é um dado primitivo, já um tipo char[ ] é um dado composto.
e) Os índices de vetores e matrizes devem ser inteiros.

Referências

DEITEL, Paul; DEITEL, Harvey. **Como programar**: em C. 6. ed. São Paulo: Bookman, 2011. 692 p.

EDELWEISS, Nina; GALANTE, Renata. **Estrutura de dados**. Porto Alegre: Bookman, 2009.

MIZRAHI, Victorine Viviane. **Treinamento em Linguagem C**. São Paulo: Pearson, 2008.

PEREIRA, Silvio do Lago. **Estrutura de Dados em C**: uma abordagem didática. São Paulo: Érica, 2016. 184p.

PINHEIRO, FRANCISCO A. C. **Elementos de programação em C**. Porto Alegre: Bookman, 2012.

SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2015.

TENENBAUM, Aaron M.; LANGSAM, Y.; MOSHE J. A. **Estruturas de dados usando C**. São Paulo: Makron Books, 1995.

Estrutura de dados

Merris Mozer

Objetivos de aprendizagem

- Analisar e distinguir as estruturas de dados relacionados à pilha, fila e listas;
- Entender a lógica utilizada para a implementação das pilhas, filas e listas;
- Conhecer e aplicar ponteiros e alocação dinâmica de memória;
- Compreender sobre os algoritmos de pesquisa;
- Compreender os conceitos relacionados à classificação.

Seção 1 | Alocação dinâmica de memória

Esta é a primeira seção da nossa Unidade 3. Nela, nós apresentamos detalhes sobre a alocação dinâmica, suas principais funções e conceitos relacionados a ponteiros. Pode-se entender a alocação dinâmica como alocação de espaços na memória em tempo de execução do programa. Essa estratégia de desenvolvimento permite que a quantidade de memória, que está sendo alocada, possa ser aumentada ou reduzida, conforme necessidade.

Seção 2 | Listas e seus casos específicos (pilha e fila)

Sendo dois dos conceitos mais usados, a pilha e a fila, ambas promovem a organização dos elementos (chegada e saída). Para a fila, consideramos o critério FIFO (do inglês *first in first out*), o primeiro item que entra é o primeiro elemento que sai; e, para a pilha, o critério LIFO (do inglês *last in, first out*), o último elemento que chega é o primeiro elemento que sai. No entanto, para implementá-los, existe uma série de estratégias, e é o que mostraremos em detalhes nesta seção.

Seção 3 | Algoritmos de pesquisa

Nesta seção, abordaremos os algoritmos para pesquisa de uma determinada informação, seja em um vetor ou matriz. Existem métodos que tornam as buscas mais eficientes; esses algoritmos, por exemplo, utilizam elementos que são usados para comparação com os demais elementos do conjunto; logo, as pesquisas podem retornar o valor procurado ou retornar nulo (caso o elemento não exista no conjunto pesquisado). Para tanto, abordaremos os seguintes métodos de pesquisa: sequencial, binário e interpolação.

Seção 4 | Classificação

Quando falamos em classificação, ou ordenação, estamos lidando com um dos ingredientes mais conhecidos na área de desenvolvimento de sistemas, cujo objetivo é organizar um conjunto de informações semelhantes em uma ordem crescente ou decrescente. Nesta seção, abordaremos suas características e seus aspectos de forma global, bem como suas funções.

Introdução à unidade

Caro(a) aluno(a), seja bem-vindo(a) à disciplina *Linguagem de Programação e Estrutura de Dados*.

Quando falamos em desenvolvimento de sistemas, precisamos pensar que a construção deve ser bem projetada, desse modo, a estrutura de dados traz os conceitos e a compreensão de como os dados devem ser armazenados e recuperados. A fim de mostrar essa importância, trazemos, nesta unidade, os conceitos relacionados à alocação dinâmica de memória, algoritmos de pesquisa, classificação, pilhas, filas e listas.

A alocação dinâmica representa um procedimento que solicita o andamento do programa e faz uso da memória dele enquanto é executado; para isso, são usados diversos métodos e conceitos, cujo detalhamento você poderá conferir na Seção 1.

Quando analisamos as aplicações existentes, dois dos conceitos mais usados é o de pilha e fila. Ambas promovem a organização dos elementos (chegada e saída). Para a fila, consideramos o critério FIFO (do inglês *first in first out*), o primeiro item que entra é o primeiro elemento que sai; para a pilha, o LIFO (do inglês *last in, first out*), o último elemento que chega é o primeiro elemento que sai. Quanto às listas, das quais pilha e fila também fazem parte, seus conceitos também serão detalhados, bem como uma série de estratégias para implementá-los – apresentados na Seção 2.

Um algoritmo de pesquisa tem como objetivo encontrar um ou mais elementos em um determinado conjunto de registros, cujo resultado, ao executá-lo, pode ser bem-sucedido ou não. Em relação aos métodos de pesquisa, existem inúmeros: **pesquisa sequencial, pesquisa binária e pesquisa interpolação**; cada um deles será descrito na Seção 3.

Por fim, na Seção 4, trataremos o processo de classificação ou, simplesmente, ordenação, que tem como objetivo organizar um conjunto de informações semelhantes em uma ordem crescente ou decrescente. Dentre os mais diversos motivos para realizar uma ordenação sequencial, pode-se ressaltar a possibilidade de acesso aos dados de forma mais eficiente.

Seção 1

Alocação dinâmica de memória

Introdução à seção

Sobre alocação dinâmica, Laureano (2008) descreve que ela ocorre em tempo de execução, na qual uma determinada variável e sua estrutura são declaradas sem que haja a necessidade de definição de tamanho. Ao executar o programa, a memória será reservada quando houver a necessidade de utilização de uma variável ou parte dela. Esse tipo de alocação é bastante usado para resoluções de problemas de estrutura de dados, como para filas, árvores dinâmicas ou listas encadeadas.

No entanto, antes de iniciar o estudo sobre as funções para alocações dinâmicas, é necessário que você entenda dois conceitos: endereços e ponteiros.

Endereços

A memória é composta por uma sequência de bytes (um byte armazena um conjunto de 256 valores); esses bytes possuem numeração sequencial e essa numeração é seu endereço.

Ponteiros

Segundo PUCRS (s.d.), um ponteiro é descrito como uma variável capaz de armazenar um endereço de memória ou o endereço de outra variável. As variáveis são posições ocupadas na memória e seus valores, normalmente, são do tipo char, int, float, double, dentre outros. Para IME2 (s.d.), uma variável do tipo ponteiro pode conter um determinado valor, ou seja, o endereço para outras posições na memória.

Para declarar um ponteiro, você deve especificar para qual tipo de variável ele irá apontar. O operador que indica a variável é o *. Exemplo: ponteiro para um inteiro, int *ponteiro.

Existem quatro tipos de funções para alocações dinâmicas, tais como: malloc(), calloc(), realloc(), sizeof() e free(); no entanto, as mais utilizadas são a malloc() e a free() – que falaremos um pouco mais a seguir.

Função malloc

Essa função tem como objetivo realizar a alocação de um bloco de byte (consecutivos) na memória RAM da máquina e fazer a devolução do endereço do bloco. Para tanto, devemos informar a quantidade de bytes para a função, e a sintaxe para a função malloc () é:

void *malloc (*valor inteiro que representa a quantidade de bytes a ser alocado*).

Na Figura 1.1 exemplificamos esse tipo de função, cuja demonstração nos revela que um determinado usuário poderá definir o tamanho do espaço na memória que deverá ser alocado.

Figura 1.1 | Exemplo função malloc()

```
#include <stdlib.h>

int heads()

        { return rand() < RAND_MAX/2; }
main(int argc, char *argv[])
        { int i, j, cnt;
            int N = atoi(argv[l]), M = atoi(argv[2]);
            int *f malloc«N+1)*sizeof(int»;
            for (j = 0; j <= N; j++) f[j] = OJ
            for (i = OJ i < M; i++, f[cnt]++)
                    for (cnt = 0, j 0; j <= N; j++)
                            if (heads(» cnt++;
            for (j = 0; j <= N; j++)
                    {
                            printf("%2d n, j);
                            for (i = 0; i < f[j] j i+=10) printf("*");
                            printf("\n")j
                    }
        }
```

Fonte: Sedgewick (1998, p. 87).

Função Free

Sempre que houver o término de uma alocação dinâmica durante a execução de um programa ou aplicação, é necessária a liberação da memória alocada. A responsabilidade dessa liberação é da função *free*.

Figura 1.2 | Exemplo função Free()

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

int main() {
        char *c; /* o ponteiro para o espaço alocado */

        /* aloco um único byte na memória */
        c = (char *)malloc(1);

        /* vejo se conseguiu alocar */
        if (!c) {
                printf("Não conseguiu alocar a memória\n");
                exit(1);
        }
        /* carrego um valor na região de memória alocada */
        *c = 'd';

        /* escrevo este valor */
        printf("%c\n",*c);

        /* libero a memória alocada */
        free(c);
}
```

Fonte: UNICAMP (s.d.).

Para saber mais

Disponibilizamos alguns materiais para complementar seu estudo.

Links

CS. The Stony Brook Algorithm Repository. Disponível em: <http://www3.cs.stonybrook.edu/~algorith/>. Acesso em: 13 ago. 2017.

IME. **Alocação dinâmica de memória**. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html>. Acesso em: 13 ago. 2017.

Livro

HERBERT, S. **C Completo e Total**. Editora Makron, 3. ed., 1997.

Questão para reflexão

Analise as características relacionadas à alocação dinâmica: quais foram as principais diferenças percebidas em relação à alocação estática? Descreva sua percepção e compartilhe com seu professor na área do aluno.

Dica de leitura sobre as diferenças entre as alocações: <https://www.inf.ufes.br/~pdcosta/ensino/2011-2-estruturas-de-dados/slides/Aula3&4%28vetores&ponteiros%29.pdf>. Acesso em: 13 ago. 2017.

Atividades de aprendizagem

1. Baseando-se na análise dos conceitos relacionados à alocação dinâmica de memória, assinale a alternativa que descreve um desses conceitos:
a) As funções free() e malloc() são muito importantes para alocação dinâmica.
b) A alocação dinâmica é feita por meio de um vetor definido inicialmente e que não tem o valor de tamanho alterado.
c) A função malloc() tem como objetivo realizar o merge de dois registros de dados.
d) Não é necessário trabalhar com ponteiros na alocação dinâmica.
e) A alocação dinâmica não é aplicável nos projetos que utilizam a metodologia ágil.

2. Analise a afirmativa: Sempre que houver o término de uma alocação dinâmica durante a execução de um programa ou aplicação, é necessária a liberação da memória alocada. Essa afirmação refere-se à qual tipo de função:
a) Função malloc ().
b) Função free ().
c) Pilha.
d) Fila.
e) Main.

3. Pode-se afirmar que existem vários tipos de funções de alocação dinâmica. Assinale a alternativa que não relacionada um tipo de alocação dinâmica:
a) Malloc( ).
b) Free ( ).
c) Int ( ).
d) Sizeof ( ).
e) Main ( ).

4. Marque a alternativa que mostra a representação gráfica de operador relacionado ao ponteiro:
a) %.
b) *.
c) #.
d) $.
e) @.

5. Analise as afirmativas relacionadas à alocação dinâmica:
I – A alocação dinâmica permite que seja definido um valor único que deve ser utilizado até o fim das operações.
II – A função malloc () representa um método de pesquisa de algoritmos.
Marque a alternativa que relaciona as afirmativas corretas:
a) Somente I está correta.
b) Somente II está correta.
c) I e II estão corretas.
d) As duas opções estão incorretas.
e) I é uma opção incompleta e não foi possível analisar e II está correta.

Fique ligado

Nesta seção, apresentamos os conceitos de alocação dinâmica: endereços, ponteiros e funções principais; bem como alguns exemplos de códigos relacionados aos conceitos apresentados.

Nas seções que estão na sequência da presente unidade, iremos avaliar algumas estruturas de dados utilizando o conceito de alocação dinâmica.

Seção 2

Listas e seus casos específicos (pilha e fila)

Introdução à seção

Ao estudarmos a disciplina de estrutura de dados, entendemos que ela é de grande importância na organização, manipulação e localização de informação em uma determinada aplicação; e três dos conceitos mais usados é o de lista, pilha e fila.

Podemos ver a lista como uma sequência de itens que estão organizados em um conjunto, não necessariamente, de maneira lógica, pode ter um endereço. Por sua vez, considerada umas das estruturas de dados mais simples, a pilha é uma das estruturas de dados mais usadas pelas equipes de desenvolvimento de software. Seu objetivo principal é acessar os itens que estão no topo da lista, aplicando o critério de que o último elemento a entrar é o primeiro a sair. Para que fique fácil o entendimento desse conceito, pense na sua pia de louça, imagine que para organizá-la você deverá colocar um prato sobre o outro, formando uma pilha, cujo último prato será o primeiro a ser lavado e a deixará.

Já no caso das filas, a estrutura e os critérios aplicados são outros, uma vez que diferem na ordem de saída dos itens, sendo que o primeiro elemento que entra é o primeiro item que sai. Fundamentalmente, apenas um item pode ser inserido no final e retirado do início. Pense que você está no mercado, faz todas as suas compras e vai para a fila do caixa; ao chegar no caixa, você nota que não tem mais ninguém, ou seja, você é o primeiro. Logo depois, chegam outros clientes, que ocupam posições seguintes à sua; você, então, paga suas compras e sai da fila - foi o primeiro a chegar e o primeiro a sair. Ficou fácil?

Esse foi apenas um resumo da seção, nos próximos tópicos vamos detalhar os conceitos de listas, pilha e fila.

Listas

Ao analisarem o conceito de lista, Tenenbaum, Langsam e Augenstein (1995) concluem que ela é composta por um endereço que faz a ligação para o próximo item, possibilitando seu acesso de forma randômica e a realização de operações, tais como, inclusão ou

exclusão. As listas podem ser lineares ou encadeadas.

a. **Listas lineares**: os elementos estão organizados de forma sequencial, embora isso não signifique estarem numa sequência física. Exemplo: você vai ao dentista, enquanto aguarda, existem várias pessoas na sala, porém, suas posições de cadeiras não estão na sequência. Assim, cada item da lista é conhecido como nó ou nodo. Dentre os exemplos de listas lineares, Tenenbaum, Langsam e Augenstein (1995) citam as pilhas e filas, cujos temas serão tratados nos próximos tópicos.
b. **Listas encadeadas**: os itens não possuem uma ordem sequencial na memória. A fim de manter-se sequencialmente lógica, podem ser codificadas de duas maneiras: simplesmente encadeada e duplamente encadeada (TENENBAUM; LANGSAM; AUGENSTEIN, 1995).

- Simplesmente encadeada: cada item tem um espaço para armazenar informação e a referência da localização na memória, considerando o item seguinte da lista;

- Duplamente encadeada: cada item possui um espaço para armazenar informação e a referência da localização na memória, considerando o item anterior da lista.

Pilha

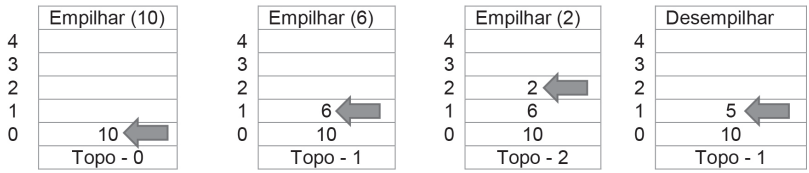
De acordo com Tenenbaum, Langsam e Augenstein (1995), pilha é um dos conceitos mais úteis, desempenhando um papel proeminente nas áreas de programação e suas linguagens. Seu conceito é descrito como um conjunto ordenado de itens, no qual novos itens podem ser inseridos e, a partir do qual, eles podem ser eliminados em uma extremidade chamada topo da pilha.

Ao analisar o contexto no qual o conceito de pilha é aplicado, identificamos: edição de textos, processo para navegação entre browsers, funções que requerem recursividade etc. Para implementar os conceitos de pilha, podem ser usados vetores ou listas encadeadas.

A principal característica de uma pilha é que a última informação a entrar é a primeira informação a sair, conhecida pela sigla LIFO (do inglês, *last in first out*). Sua estrutura é composta pelos seguintes métodos: push (empilhar informação), pop (desempilhar informação), size (retornar o tamanho total da pilha), *stackpop* (retornar maior elemento sem que seja removido) e *empty* (verificação se a pilha

está vazia). Observe a Figura 1.3, nela, exemplificamos a entrada de informação, como são posicionadas em um vetor e, depois, quando uma informação foi desempilhada, qual foi o posicionamento do topo e qual informação saiu do vetor.

Figura 1.3 | Exemplo de pilha



Fonte: elaborada pelo autor.

A implementação de uma pilha pode ocorrer de duas formas: por meio de vetor ou ponteiros.

Implementação por vetor

De acordo com Tenenbaum, Langsam, Augenstein (1995), a implementação de uma pilha por meio de um vetor envolve a declaração de dois objetos, um vetor para armazenamento dos itens da pilha e um inteiro, indicando a atual posição do topo no vetor. Em seguida, podem ser aplicados os métodos/as funções de pilha e outras condições que forem necessárias. Observe a sintaxe do algoritmo abaixo (IME, [s.d.]).

Figura 1.4 | Algoritmo de pilha

```
#define N 100
char pilha[N];
int t;

// Esta função devolve 1 se a string s contém uma
// sequência bem-formada de parênteses e colchetes
// e devolve 0 se a sequência é malformada.

int bemFormada (char s[]) {
  criapilha ();

  for (int i = 0; s[i] != '\0'; ++i) {
    char c;
    switch (s[i]) {
      case ')': if (pilhavazia ()) return 0;
                c = desempilha ();
                if (c != '(') return 0;
                break;
      case ']': if (pilhavazia ()) return 0;
                c = desempilha ();
                if (c != '[') return 0;
                break;
      default:  empilha (s[i]);
    }
  }
  return pilhavazia ();
}

void criapilha (void) {
  t = 0;
}
void empilha (char y) {
  pilha[t++] = y;
}

char desempilha (void) {
  return pilha[--t];
}

int pilhavazia (void) {
  return t <= 0;
}
```

Fonte: IME (s.d.).

Implementação por Ponteiros

Para a implementação de pilha usando ponteiros, Bertol (s.d.) cita o seguinte exemplo de algoritmo e detalha seus comentários no código:

Figura 1.5 | Algoritmo de pilha por ponteiro

```
#include "stdio.h"
#include "conio.h"
#include "string.h"
#include "stdlib.h"

// modelo matemático (estrutura de dados)
struct TipoItem {     // cada item da pilha corresponde a um
  char nome[30];      // registro (TipoItem) composto apenas
};                    // do campo nome

typedef struct Celula *Apontador;          // define o tipo "Apontador" como sendo o
                                           // endereço de uma "Celula"
struct Celula {
    TipoItem Item;
    Apontador prox;
  };

struct TipoPilha {
  Apontador Topo;
};

// conjunto de operações que podem ser aplicadas sobre o modelo 'TipoPilha'
void FazPilhaVazia(TipoPilha *Pilha);
int PilhaVazia(TipoPilha *Pilha);
void Empilha(TipoItem x, TipoPilha *Pilha);
int Desempilha(TipoPilha *Pilha, TipoItem *x);
void ImprimePilha(TipoPilha *Pilha);

void main() {
  TipoItem x;
  TipoPilha Pilha;
  FazPilhaVazia(&Pilha); // faz a Pilha ficar vazia
  while (1) {
    ImprimePilha(&Pilha);
    printf("\nInforme um nome do item a ser empilhado, (FIM) para encerrar:\n");
    gets(x.nome);
    if (strcmp(x.nome, "FIM") == 0)
      break;
    Empilha(x, &Pilha);
```

```
}
// Faz a 'Pilha' ficar vazia criando a célula cabeça
void FazPilhaVazia(TipoPilha *Pilha) {
  Pilha->Topo = (Apontador) malloc(sizeof(Celula));
  Pilha->Topo->prox = NULL;
}

// Esta função retorna 1 (true) se a 'Pilha' está vazia; senão retorna 0 (false)
int PilhaVazia(TipoPilha *Pilha) {
  return(Pilha->Topo->prox == NULL);
}

// Insere o item 'x' no 'Topo' da 'Pilha'.
void Empilha(TipoItem x, TipoPilha *Pilha) {
  Apontador p;
  p = (Apontador) malloc(sizeof(Celula)); // cria uma nova célula cabeça
  Pilha->Topo->Item = x; // coloca o item "x" na antiga célula cabeça
// atualiza o topo da pilha
  p->prox = Pilha->Topo;
  Pilha->Topo = p;
}

// Retira o item 'x' que está no topo da 'Pilha'
int Desempilha(TipoPilha *Pilha, TipoItem *x) {
  if (PilhaVazia(Pilha))
     return(0); // Erro: Pilha vazia.
  else {
    Apontador p;
    p = Pilha->Topo;
    Pilha->Topo = Pilha->Topo->prox;
    *x = Pilha->Topo->Item; // item retornado
    free(p);
    return(1); // Item retirado com sucesso
  }
}

void ImprimePilha(TipoPilha *Pilha) {
  TipoItem x;
  TipoPilha PilhaAux;
  FazPilhaVazia(&PilhaAux);
  clrscr();
  while (!PilhaVazia(Pilha)) {
    Desempilha(Pilha, &x);
    printf("%s\n", x.nome);
    Empilha(x, &PilhaAux);        // salva os itens desempilhados da 'Pilha'
  }                               // na 'PilhaAux'

// retorna o itens para a pilha original (Pilha)
  while (!PilhaVazia(&PilhaAux)) {
    Desempilha(&PilhaAux, &x);
    Empilha(x, Pilha);
  }
}
```

Fonte: Bertol (s.d.).

Fila

A fila representa um conjunto com itens ordenados; a partir desse conjunto é possível executar a eliminação dos itens que estão em

uma das extremidades (início da fila), e são adicionados ou deletados seguindo o conceito de que o primeiro que entra é o primeiro que sai (do inglês, FIFO – *first in*, *first out*, traduzido para o português como o primeiro que entra é o primeiro que sai). Esse conceito não é apenas aplicável à área de desenvolvimento de sistemas, preste atenção no seu dia a dia, as filas estão presentes no supermercado, nas instituições bancárias ou na hora de pagar a pipoca no cinema. Em um sistema, podemos ter esse conceito aplicado para mensagens trocadas em uma rede, controlar fila de impressão de documentos etc.

Os métodos básicos usados para a fila são: *insert* (inserir novos itens em uma fila - no final), *remove* (excluir o item da fila – no início), *empty* (verificação se a fila está vazia), *size* (retornar o tamanho da fila) e front (retornar o item na sequência, sem que seja retirado).

A implementação de uma fila pode ocorrer de duas formas: por meio de vetor ou ponteiros.

Implementação por vetor

Para implementar uma fila por vetor, estrutura a fila com o número de itens (n), um vetor para armazenamento dos itens e um inteiro para determinar a posição atual do vetor que armazena o primeiro item da fila. No exemplo citado a seguir, podemos ver o algoritmo de cálculo de distância entre cidades, identificando a fila ordenada de interligações:

Figura 1.6 | Algoritmo de fila por vetor

```
#define N 100
int fila[N], int p, u;
int dist[N];

void criafila (void) {
  p = 0; u = 0;
}

int filavazia (void) {
  return p >= u;
}

int tiradafila (void) {
  return fila[p++];
}

void colocanafila (int y) {
  fila[u++] = y;
}

void distancias (int A[][N], int c) {
  for (int j = 0; j < N; ++j)  dist[j] = N;
  dist[c] = 0;
  criafila ();
  colocanafila (c);

  while (!filavazia ()) {
    int i = tiradafila ();
    for (int j = 0; j < N; ++j)
      if (A[i][j] == 1 && dist[j] >= N) {
        dist[j] = dist[i] + 1;
        colocanafila (j);
      }
  }
}
```

Fonte: IME (s.d.).

Implementação por Ponteiro

Mesmo com a possibilidade de implementação da fila por meio de vetores, a utilização de ponteiros, para torná-la dinâmica, torna-se uma boa prática, visto que ela pode se expandir. Esse exemplo mostra uma fila que retorna os números reais:

Figura 1.7 | Algoritmo de fila por ponteiro

```
#include
struct Fila {

		int capacidade;
		float *dados;
		int primeiro;
		int ultimo;
		int nItens;
}

void criarFila( struct Fila *f, int c ) {
		f->capacidade = c;
		f->dados = (float*) malloc (f->capacidade * sizeof(float));
		f->primeiro = 0;
		f->ultimo = -1;
		f->nItens = 0;
}
void inserir(struct Fila *f, int v) {

		if(f->ultimo == f->capacidade-1)
			f->ultimo = -1;

		f->ultimo++;
		f->dados[f->ultimo] = v; // incrementa ultimo e insere
		f->nItens++; // mais um item inserido

}

int remover( struct Fila *f ) { // pega o item do começo da fila

		int temp = f->dados[f->primeiro++]; // pega o valor e incrementa o primeiro

		if(f->primeiro == f->capacidade)
			f->primeiro = 0;

		f->nItens--;  // um item retirado
		return temp;

}
```

```
int estaVazia( struct Fila *f ) { // retorna verdadeiro se a fila está vazia

		return (f->nItens==0);

}

int estaCheia( struct Fila *f ) { // retorna verdadeiro se a fila está cheia

		return (f->nItens == f->capacidade);
}

void mostrarFila(struct Fila *f){

		int cont, i;

		for ( cont=0, i= f->primeiro; cont < f->nItens; cont++){

				printf("%.2f\t",f->dados[i++]);

				if (i == f->capacidade)
					i=0;

		}
		printf("\n\n");
}
void main () {

		int opcao, capa;
		float valor;
		struct Fila umaFila;

		// cria a fila
		printf("\nCapacidade da fila? ");
		scanf("%d",&capa);
		criarFila(&umaFila, capa);

		// apresenta menu
		while( 1 ){

				printf("\n1 - Inserir elemento\n2 - Remover elemento\n3 - Mostrar Fila\n0 -
Sair\n\nOpcao? ");
				scanf("%d", &opcao);

				switch(opcao){

					case 0: exit(0);

					case 1: // insere elemento
						if (estaCheia(&umaFila)){

							printf("\nFila Cheia!!!\n\n");

						}
						else {

							printf("\nValor do elemento a ser inserido? ");
							scanf("%f", &valor);
							inserir(&umaFila,valor);

						}

						break;
```

```
                        case 2: // remove elemento
                                if (estaVazia(&umaFila)){

                                        printf("\nFila vazia!!!\n\n");
                                }
                                else {
                                        valor = remover(&umaFila);
                                        printf("\n%1f removido com sucesso\n\n",
valor) ;
                                }
                                break;

                                case 3: // mostrar fila
                                        if (estaVazia(&umaFila)){

                                                printf("\nFila vazia!!!\n\n");

                                        }
                                        else {

                                                printf("\nConteudo da fila => ");
                                                mostrarFila(&umaFila);
                                        }
                                        break;

                                default:
                                        printf("\nOpcao Invalida\n\n");
                        }
                }
}
```

Fonte: UFRJ (s.d.).

Questão para reflexão

Pesquise sobre o desempenho de filas, pilhas e listas; identifique qual delas tem melhor desempenho em uma aplicação e compartilhe com seu professor.

Para saber mais

Disponibilizaremos alguns materiais que complementarão o estudo dos temas desta unidade.

Links

UNICAMP. **Apostilas**. Disponível em: <http://www.ic.unicamp.br/~ra069320/PED/MC102/1s2008/Apostilas/>. Acesso em: 13 ago. 2017.

UFMG. **Estrutura de dados básica**. Disponível em: < http://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/lists.pdf>. Acesso em: 14 ago. 2017.

Livros

ZIVIANI, N. **Projeto de Algoritmos**. 2. ed., Editora Thomson.

SEDGEWICK, R. **Algorithms in C**. 3. ed., Editora Addison-Wesley, 2008.

Atividades de aprendizagem

1. Em relação aos conceitos relacionados às listas, analise as opções abaixo:
I - ____ pode-se concluir que ela é composta por um endereço que faz a ligação para o próximo item, possibilitando seu acesso de forma randômica e a realização de operações, tais como inclusão ou exclusão.
II -____. Seu conceito é descrito como um conjunto ordenado de itens no qual novos itens podem ser inseridos.
Marque a alternativa que preenche, respectivamente, as lacunas:
a) Lista e Pilha.
b) Lista e Fila.
c) Fila e Pilha.
d) Fila e Lista.
e) Malloc e Free.

2. Em relação aos conceitos relacionados às listas, analise as opções abaixo:
I - ____ pode-se concluir que ela é composta por um endereço que faz a ligação para o próximo item, possibilitando seu acesso de forma randômica e a realização de operações, tais como, inclusão ou exclusão.
II -____. Seu conceito é descrito como um conjunto ordenado de itens no qual novos itens podem ser inseridos.
Marque a alternativa que preenche, respectivamente, as lacunas:
a) Lista e Pilha.
b) Lista e Fila.
c) Fila e Pilha.
d) Fila e Lista.
e) Malloc e Free.

3. Analise as afirmativas:
I – Na Pilha é possível retirar o item que está na última posição.
II – Na Fila é possível ter um item que está na última posição e será o primeiro a sair.

Assinale a alternativa que apresenta a resposta correta:
a) I está correta.
b) II está correta.
c) I e II estão corretas.
d) I e II estão incorretas.
e) I está correta e II está incorreta.

4. Analisando as características relacionadas à fila, assinale a opção que não representa um dos métodos básicos usados por ela:
a) *insert*.
b) *remove*.
c) *size*.
d) *time*.
e) *while*.

5. Analise a descrição: sua estrutura é composta pelos seguintes métodos: push, pop, size, stackpop e empty. Essa descrição está relacionada à:
a) Implementação por vetor de uma lista.
b) Fila.
c) Pilha.
d) Lista.
e) Classes.

Fique ligado

Nesta seção, compartilhamos os conceitos sobre fila, pilha e lista, mostrando a importância da organização, manipulação e localização de uma informação em específico e percebendo que esses conceitos são amplamente difundidos na área de desenvolvimento de aplicações.

Seção 3

Algoritmos de pesquisa

Introdução à seção

A presente seção tem como objetivo apresentar os algoritmos de pesquisa para busca de uma determinada informação, seja em um vetor ou uma matriz. O raciocínio para elaboração de um algoritmo de busca é baseado na comparação entre o elemento a ser procurado e cada um dos elementos que pertencem ao vetor ou matriz. Essa comparação é executada até que o elemento em questão seja encontrado ou que, após uma varredura completa, seja identificado que ele não pertence ao conjunto pesquisado.

Para Tenenbaum, Langsam, Augenstein (1995), o algoritmo de busca deve aceitar um argumento e buscá-lo em um conjunto de elementos. O retorno do elemento pode ser inteiro ou ponteiro, uma vez que tenha sido encontrado ou não.

Dentre os fatores que influenciam o desempenho de uma pesquisa, podemos citar a forma com que os elementos estão organizados: ordenados ou desordenados. No primeiro caso, existe a necessidade de verificação do primeiro ao último elemento do vetor ou matriz. No segundo caso, ao compararmos o elemento buscado com um elemento do vetor ou matriz e o identificarmos como maior ou menor, pode-se chegar a uma conclusão sobre sua inexistência.

Assim, considerando que a atividade de pesquisa de dados é uma atividade habitual e que exige algoritmos eficientes em desempenho, nos tópicos a seguir, apresentaremos os seguintes métodos de pesquisas: **pesquisa sequencial, pesquisa binária e pesquisa interpolação**.

a. Pesquisa sequencial

Para Tenenbaum, Langsam, Augenstein (1995), a pesquisa sequencial ou pesquisa linear é considerada o método mais simples de pesquisa, a qual pode ser descrita como análise de todos os elementos do vetor de forma sistemática. Essa análise inicia no primeiro elemento do vetor e prossegue para os seguintes até que encontre o elemento procurado ou até finalizar o conjunto de elementos. Considerando essa característica, a pesquisa sequencial é um método demorado e dependente do tamanho total do vetor.

Para entender qual é o comportamento de uma pesquisa sequencial, analise o algoritmo demonstrado na Figura 1.8:

Figura 1.8 | Exemplo algoritmo pesquisa sequencial

```
// Retornar um valor inteiro x em um vetor
// crescente v[0..n-1] e retorna um índice j
// em 0..n tal que v[j-1] < x <= v[j].

int pesquisaSequencial (int x, int n, int v[]) {
   int j = 0;
   while (j < n && v[j] < x)
      j++;
   return j;
}
```

Fonte: elaborada pelo autor.

Após uma análise, existem quantos loops comparativos, entre X e os elementos pertencentes ao vetor? Podemos concluir que existirão n comparações. Se o tamanho do vetor for 10, por exemplo, a quantidade de loops de comparação Serão multiplicada por 10 e o tempo despendido para a pesquisa levará em consideração o número de comparações realizadas.

Dentre suas vantagens, podemos citar: forma mais simples de busca, melhor eficiência para quantidade pequena e média de informações e todos os elementos do conjunto podem ser pesquisados. Em relação às desvantagens, estão: a tabela precisa estar ordenada, existe espaço adicional para armazenamento de índices de pesquisa e baixa eficiência para grande volume de dados.

b. Pesquisa binária

Considerando a necessidade de aceleração dos métodos de pesquisa, uma estratégia a ser utilizada é o particionamento de forma sucessiva do conjunto de valores do vetor, a fim de reduzir a quantidade de elementos a serem analisados. No entanto, esse é um método de pesquisa que apenas funcionará se o conjunto de elementos estiver ordenado.

Vamos ao seguinte exemplo: você está pesquisando o elemento **5** em um dado conjunto de valores; esse conjunto de valores é igual a 1,2,3,**4**,5,6,7. A pesquisa binária irá analisar o elemento que está no meio do conjunto, no vetor citado, o valor **4**. Ao comparar o elemento pesquisado com o elemento de valor médio, é verificado

que o elemento pesquisado **5** é maior do que o elemento médio, a pesquisa continuará na segunda metade do vetor e a primeira metade é descartada. Agora, na segunda metade do vetor (5,6 e 7), o elemento médio é o **6**, que é maior que 5, então 6 e 7 são descartados. Assim, o elemento é encontrado.

Observe o algoritmo da Figura 1.9. Neste algoritmo, a proposta foi a busca por um determinado elemento X que percorre o vetor e retorna -1, pelo fato do elemento não ser encontrado.

Figura 1.9 | Exemplo de algoritmo pesquisa binária

```
int
buscaBinaria (int x, int n, int v[]) {
  int e, m, d;                    // 1
  e = 0; d = n-1;                 // 2
  while (e <= d) {                // 3
    m = (e + d)/2;                // 4
    if (v[m] == x) return m;      // 5
    if (v[m] < x) e = m + 1;      // 6
    else d = m - 1;               // 7
  }                               // 8
  return -1;                      // 9
}
```

Fonte: IME (s.d.).

c. Pesquisa interpolação

De acordo com Tenenbaum, Langsam, Augenstein (1995), a pesquisa por interpolação é outro método a ser aplicado em um vetor ordenado, tornando-se uma variante melhorada da pesquisa binária. Inicialmente, *low* é definido com 0 e high torna-se n-1, e, no algoritmo, a chave de argumento *key* será reconhecida por estar entre low e *high*. Considerando que as chaves estão uniformemente distribuídas entre esse intervalo de valores, esperamos que a *key* esteja em posição aproximada: *mid = low +(high - low) * ({key - k(low))/k(high) - k(low)))* (TENENBAUM, 1995).

Ao aplicar a fórmula de aproximação, se a *key< mid*, é necessário redefinir high como mid-1, caso contrário, faça a redefinição de low como mid+1. Esse processo é necessário até que a chave seja encontrada ou que *low> high* (TENENBAUM, 1995).

É importante ressaltar que a busca por interpolação é lenta, pois envolve cálculos aritméticos sobre as chaves, além de complexas multiplicações e divisões, tornando-a mais vagarosa que a busca binária, mesmo que o processo de comparações seja com menos iterações

(TENENBAUM, LANGSAM, AUGENSTEIN, 1995). A representação dessa busca em forma de algoritmo segue na Figura 1.10, logo abaixo:

Figura 1.10 | Exemplo algoritmo pesquisa por interpolação

```
int pesqInter(int chave, int v[], int n) {
        int ini = 0;
        int meio;
        int fim = n - 1;
        while (ini <= fim) {
                meio = ini + ((fim-ini)*(chave-v[ini])) / (v[fim]-v[ini]);
                printf("\n O indice do meio foi: %i", meio);
                if (chave < v[meio]) {
                        fim = meio - 1;
                } else if (chave > v[meio]) {
                        ini = meio + 1;
                } else {
                        return meio;
                }
        }
        return -1; // Índice Impossível
}
```

Fonte: UNICAMP (s.d.).

Questão para reflexão

Analise o seguinte cenário: Carlos é gerente de projeto em uma grande empresa de construção civil e está responsável por gerenciar a execução do mais novo projeto de casas de condomínios fechados na sua cidade. A empresa possui um sistema que gerencia todo o ciclo de vida dos projetos e todas as informações pertinentes. Analise opções de como os algoritmos de pesquisa podem ser utilizados no sistema de gestão de projetos usado pelo Carlos. Compartilhe sua opinião com o professor.

Dica: <http://pmbook.ce.cmu.edu/10_Fundamental_Scheduling_Procedures.html> (Acesso em: 28 de jul. 2017).

Para saber mais

Complementaremos seu material de estudo com alguns links sobre algoritmos.

Links

ALGOSORT. **Computer Programming Algorithms Directory**. Disponível em: <http://www.algosort.com/>. Acesso em: 28 jul. 2017.

PRINCETON. **Algorithms**. Disponível em: <http://algs4.cs.princeton.edu/lectures/13StacksAndQueues.pdf>. Acesso em: 28 jul. 2017.

MIT. **Introduction to Algorithms (SMA 5503)**. Disponível em: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/>. Acesso em: 28 jul. 2017.

ZUNNY. **Using the "bestfast" search algorithm and "profile" tables**. Disponível em: <http://zunny.com/RUBIK.HTM>. Acesso em: 28 jul. 2017.

Atividades de aprendizagem

1. Analise a sequência de números apresentados no vetor representado abaixo:

5	10	15	20	25	30	35	40	45
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

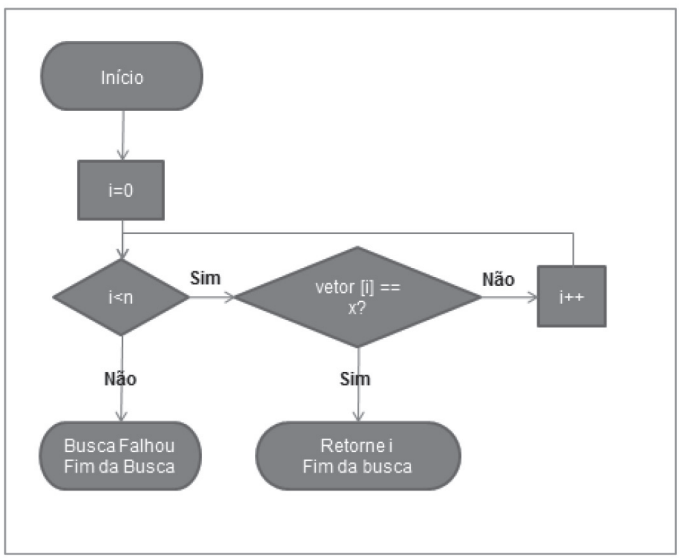
Utilizando um algoritmo de pesquisa binária, quantos loops serão necessários até que o elemento 45 seja encontrado?
a) 8.
b) 5.
c) 3.
d) 2.
e) 1.

2. Esse método utiliza um vetor ordenado e realiza particionamento do espaço de busca, realizando a comparação do elemento a ser localizado com o elemento no meio do vetor. Caso o elemento a ser localizado seja igual ao elemento do meio, a pesquisa é encerrada. Senão, a pesquisa continua o particionamento, até o que o elemento seja localizado ou todos os elementos sejam pesquisados e a busca encerrada. Baseando-se nessa descrição, de qual método estamos falando?
a) Pesquisa sequencial.
b) Pesquisa binária.

c) Pesquisa sequencial recursiva.
d) A descrição não está relacionada a nenhum tipo de pesquisa.
e) A descrição pode ser usada para todos os tipos de pesquisa.

3. João é um professor de Lógica e citou a seguinte frase: Para resolver um problema, precisamos dividi-lo em pequenas partes menores; a partir dessa divisão, podemos ver que ele ficará mais simples.
Ao fazer uma analogia com os métodos de pesquisa, estamos nos referindo à:
a) Pesquisa sequencial.
b) Pesquisa binária.
c) Pesquisa por interpolação.
d) Nenhuma das alternativas anteriores.

4. Analise a seguinte imagem:



Utilizando a representação gráfica da figura, verifique quantas iterações serão necessárias até encontrar o valor 8 no vetor:

1	2	3	7	8	15	19

a) 5.
b) 8.
c) 2.
d) 3.
e) 5.

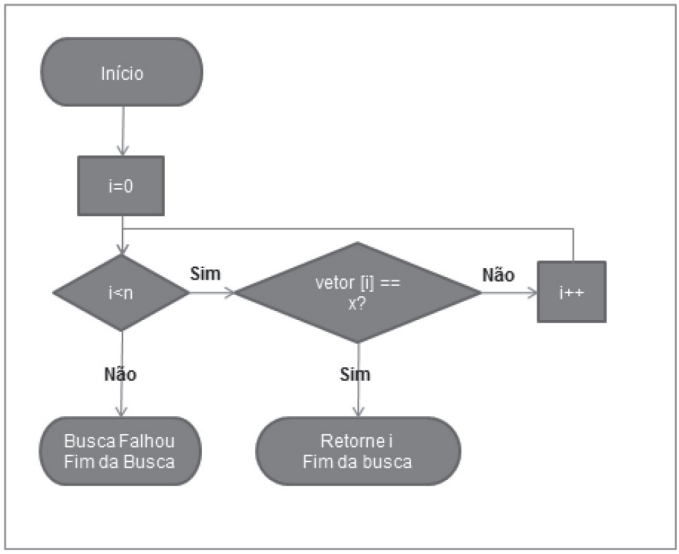
5. Analise as opções a seguir:
I – Pesquisa sequencial: existem cálculos aritméticos para localizar o valor buscado.
II – Pesquisa binária: se o vetor contém 8 posições e o valor buscado está na 8ª posição, consequentemente, o total de verificações sequenciais será igual a 8.
É correto afirmar que:
a) I está correta e II está incorreta.
b) I e II estão corretas.
c) I está incorreta e II está correta.
d) I e II estão incorretas.

Fique ligado

A pesquisa de dados é a base fundamental na área de desenvolvimento de software, garantindo que dados importantes sejam recuperados para utilização no cotidiano das mais variadas instituições públicas ou privadas. Dada essa importância, faz-se necessária a projeção de algoritmos que sejam confiáveis e eficientes no retorno de dados. Para tal, existem métodos de pesquisa; abaixo verifique as figuras que representam graficamente esses métodos.

e. Pesquisa sequencial:

Figura 1.11 | Representação gráfica - pesquisa sequencial



Fonte: elaborada pelo autor.

Legenda:

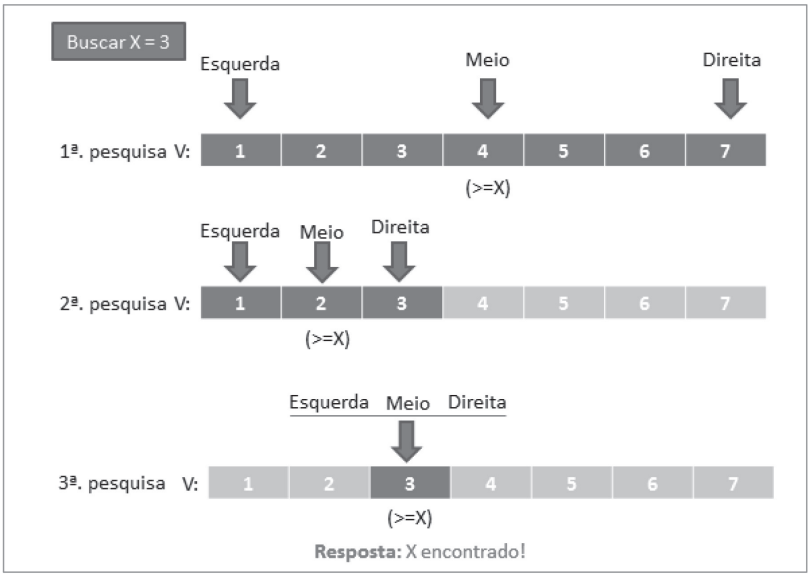
i = posição do vetor

n = tamanho do vetor

x= elemento a ser localizado no vetor

f. Pesquisa binária:

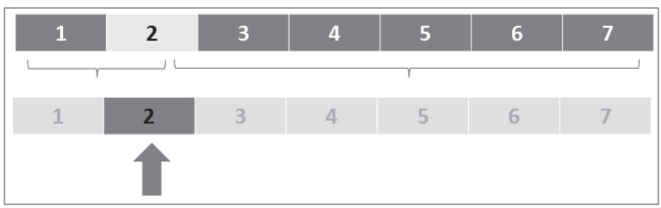
Figura 1.12 | Representação gráfica - pesquisa binária



Fonte: elaborada pelo autor.

g. Pesquisa por interpolação:

Figura 1.13 | Representação gráfica - pesquisa por interpolação



Fonte: elaborada pelo autor.

Seção 4

Classificação

Introdução à seção

Ao citar o conceito de um determinado conjunto ordenado é perceptível que ele tem um impacto na nossa rotina. Pense na seguinte cena: você vai até a biblioteca da nossa instituição para realizar um empréstimo de um livro de estrutura de dados. Ao chegar lá, verifica que existem muitas prateleiras, mas, como todos os livros fazem parte de um catálogo, todos estão em suas posições específicas, logo, você realizará a consulta da posição do livro buscado e será direcionado à prateleira correta. Assim, como no nosso exemplo, de forma geral, um determinado conjunto de itens recebe uma classificação para produção de relatórios ou para que o procedimento de acesso aos dados seja mais eficiente.

Podemos dividir a classificação em dois grupos: interna, quando o conjunto de dados está contido na memória principal, e externa, quando seu armazenamento não está na memória. Em relação aos métodos de classificação interna, estão: inserção, troca, seleção, intercalação e distribuição.

Classificação interna por bolha (*bubble sort*)

Provavelmente, é o tipo de classificação mais conhecido; uma das suas principais características é a facilidade de entendimento e programação; no entanto, é a menos eficiente, pois a ideia central é percorrer o arquivo de forma sequencial diversas vezes; a cada iteração, um item é comparado com o seu sucessor e trocados de ordem, caso necessário. O algoritmo que representa esse método está demonstrado na Figura 1.14:

Figura 1.14 | Exemplo de método bolha

```
bubble (x, n)
int x[], n;
 {
 int hold, j, pass;
 int switched = TRUE;
 for (pass = 0; pass < n-1 && switched == TRUE; pass++) {
 /* repeticao mais externa controla nº de passagens */
          switched = FALSE; /* inicialmente nenhuma troca */ /* foi feita nesta passagem */
          for (j = 0; j < n-pass-1; j++)
           /* repeticao mais interna controla cada pass indiv */
                    if (x[j ] > x[j+l ) {
                              /* elementos fora de ordem */ l* eh necessária uma troca */
                             switched = TRUE;
                             hold = x[j] ;
                              x[j] = x[j+l];
                             x[j+l] = hold;
                    } / * fim if */
          } / fim for * /
 } /* fim bubble */
```

Fonte: Tenenbaum (1995).

Classificação interna por troca de partição (*quicksort*)

Este método consiste em um algoritmo que particiona o vetor e permite que um valor específico seja alocado na posição correta. O algoritmo que representa esse método é:

Figura 1.15 - Exemplo de método *quicksort*

```
partition (x, lb, ub, pj)
int x[ ], lb, ub, *pj;
 {
 int a, down, temp, up;
 a = x[lb]; /* a eh o elemento cuja posicao */ /* final eh procurada */
 up = ub;
 down = lb;
 while (down < up) {
          while (x[down] <= a && down < ub)
                    down++; /* sobe no vetor */
          while (x[up] > a)
                    up--; /* desce no vetor */
           if (down < up) {
                    /* troca x[down] e x[up] */
                    temp = x[down];
                    x[down] = x[up];
                    x[up] = temp;
          } /* fim if */
} /* fim while */
          x[lb] = x[up];
          x[up] = a;
          pj = up;
 } /* fim partition */
```

Fonte: TENENBAUM (1995).

Classificação interna por seleção

Este método é aquele no qual sucessivos itens são selecionados sequencialmente e organizados em suas posições de forma ordenada. Consiste em trocar o menor item de uma determinada lista com o elemento posicionado no início da lista, em seguida, o segundo menor item com a segunda posição e assim sucessivamente com os (n - 1) itens restantes. Observe nosso exemplo de método de seleção:

Figura 1.16 | Exemplo de método seleção

```
define dpq com a fila de prioridade descendente vazia;
 /* pre-processa os elementos do vetor de entrada */
 /* inserindo-os na fila de prioridade */
 f o r (i=0; i<n; i++)
          pqinsert(dpq, x[i]);
          /* seleciona cada elemento sucessivo em sequência */
          for (i - n - 1; i >= 0; 1--)
                    x[i] = pqmaxdelete(dpq);
```

Fonte: Tenenbaum (1995).

Classificação interna por inserção

Uma ordenação por inserção refere-se à ordenação de um conjunto de registros inserindo itens num arquivo ordenado já existente. Observe o algoritmo a seguir.

Figura 1.17 | Exemplo de método inserção

```
insertsort(x, n)
 int x[], n;
{ int i, k, y;
/* Inicialmente x[0] é considerado um arq classif de  */
*/ I* um elemento. Apos cada interação,*/
 /* os elementos x[0] a x[k] estarão em sequencia. */
          for (k =1; k<n; k++) {
                    /*Move 1 posição p/ baixo todos elems maiores que y*/

                    for (k =1; k= 0 && y < x[i] ; i—)
                              x[i+l] = x[i] ; /* Insere y na posição correta */
                    x[i+l] = y;
                    } /* fim for */
} I* fim insertsort */
```

Fonte: Tenenbaum (1995).

Classificações por intercalação

Este processo combina dois ou mais arquivos classificados num terceiro arquivo classificado. Confira o exemplo na Figura 1.18:

Figura 1.18 | Exemplo de método intercalação

```
mergearr(a , b, c, nl , n2, n3)
i n t a[] , b[] , c[] , nl , n2, n3 ;
 {
        int apoint, bpoint, cpoint;
        int alimit; blimit, climit;

        alimit = nl-1;
        blimit = n2-l;
        climit = n3-l;
        if (nl+n2 != n3) {
                printf("os tamanhos dos vetores sao incompativeis/n");
                exit(l);
        } /* fim if * l /
        /* apoint e bpoint indicam a posicao em que nos */
        /* l* encontramos dentro dos vetores a e b respectivamente*/
        apoint = 0;
        bpoint = 0;
        for (cpoint =0 ; apoint <= alimit && bpoint <= blimit; cpoint++)
                if(a[apoint]< b[bpoint])
                        c[cpoint] = a[apoint++];
                else c[cpoint] = b[bpoint++];
        while (apoint <= alimit)
                c[cpoint++] = a[apoint++];
        while (bpoint<= blimit)
                c[cpoint++] = b[bpoint++];
 } /* fim mergearr */
```

Fonte: Tenenbaum (1995).

Questão para reflexão

Analise os métodos de classificação e descreva, na sua opinião, qual dos métodos apresenta um melhor desempenho? Desenvolva sua percepção e compartilhe no portal do aluno.

Para saber mais

Nesta seção apresentaremos alguns material complementar de apoio aos seus estudos de Ordenação.

Links

<http://www.ufpa.br/sampaio/curso_de_estdados_2/jota_jota_ordenacao/Intercalacao_de_arquivos.htm>

<https://www.researchgate.net/profile/Gf_Cintra/publication/279708678_Pesquisa_e_Ordenacao_de_Dados/links/5597f0e908ae793d137dfa16.pdf>.

<http://www.lbd.dcc.ufmg.br/colecoes/sbac-pad/1987/0039.pdf>.

<https://books.google.com.br/books?hl=pt-BR&lr=&id=DjyTjonm01sC&oi=fnd&pg=PA1&dq=Classifica%C3%A7%C3%B5es+por+intercala%C3%A7%C3%A3o+e+de+raiz&ots=uC8ml3NO86&sig=64YrwiXtScLMWgBeNUrSJz9ZEN4#v=onepage&q&f=false>.

Atividades de aprendizagem

1. Tipo de classificação mais conhecido, uma das suas principais características é a facilidade de entendimento e programação; no entanto, é a menos eficiente, pois a ideia centrar é percorrer o arquivo de forma sequencial diversas vezes. Essa descrição está relacionada à qual método de classificação:

a) Bolha.
b) Inserção.
c) Seleção.
d) Quicksort.
e) Fila.

2. O _____ é um método que consiste em um algoritmo que particiona o vetor e permite que um valor específico seja alocado na posição correta. Marque a alternativa que preenche a lacuna:

a) Bolha.
b) Inserção.
c) Seleção.
d) Quicksort.
e) Lista.

3. O trecho do algoritmo abaixo refere-se à qual método de classificação?

```
f o r (i=0; i<n; i++)
        pqinsert(dpq, x[i]);
        /* seleciona cada elemento sucessivo em sequência */
        for (i - n - 1; i >= 0; 1--)
x[i] = pqmaxdelete(dpq);
```

a) Inserção.
b) Bolha.
c) Seleção.
d) Quicksort.
e) Lista.

Fique ligado

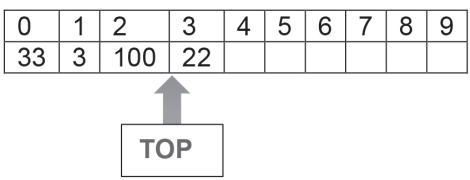
Chegamos ao final da unidade, você percebeu que a estrutura de dados apresenta inúmeros benefícios, tais como, organização da informação, melhoria de desempenho, reutilização de códigos, dentre outros. Pesquise e analise uma aplicabilidade da estrutura de dados na área de Gestão de Projetos, identificando as principais características dos métodos usados e de que forma apoia o planejamento e a identificação de possíveis desvios na sua execução. *Compartilhe o resultado de sua análise com o professor!*

Para concluir o estudo da unidade

Quando falamos em desenvolvimento de software, devemos pensar nos métodos que serão utilizados para implementá-lo; assim, o produto final deverá ter sua estrutura de dados arquitetada e, portanto, seus dados organizados; custo reduzido, tanto para criação quanto para manutenção; reutilização de códigos, além de proporcionar a interoperabilidade e bom desempenho.

Atividades de aprendizagem da unidade

1. Abaixo, temos uma pilha, observe:



Fonte: elaborada pelo autor.

Sabemos que é uma pilha, pois a estrutura acima possui um único apontador denominado TOPO.

Assim, na operação de inserção de pilha, analise a sequência de inserções a seguir:

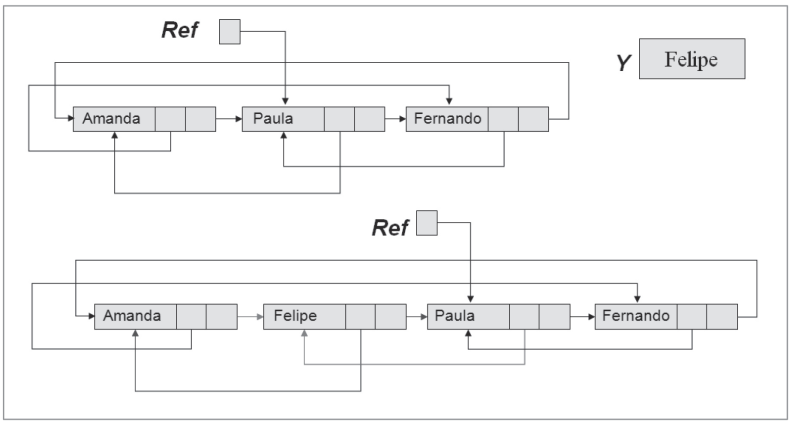
I. Inserir o elemento 33, topo ocupa posição zero.
II. Inserir o elemento 3, topo ocupa posição 1.
III. Inserir o elemento 100, topo ocupa posição 2.
IV. Inserir o elemento 22, topo ocupa posição 3.
Assinale a alternativa correta:
a) As alternativas I, II, III e IV estão corretas.
b) As alternativas I, III e IV estão corretas.
c) A alternativa I está correta e a alternativa II está errada.
d) A alternativa I está correta e a alternativa III está errada.
e) As alternativas I e II estão corretas e as alternativas III e IV estão erradas.

2. Um navegador, conhecido na informática por browser, é um programa que nos permite acessar a internet; ou seja, permite que o usuário "navegue" na rede mundial de computadores. Muitos desses sites possuem links para outros sites, e é neste momento de navegação que, através do link, pode-se abrir uma outra página, e a partir dessa página acessar outra.
Diante do exposto acima o Navegador trabalha com:
a) Uma estrutura de Lista Estática Linear com disciplina de FILA.
b) Uma estrutura de Lista Estática Linear com disciplina de PILHA.
c) Uma estrutura de Lista Encadeada com disciplina de FILA.
d) Uma estrutura de Lista Encadeada com disciplina de PILHA.
e) Uma estrutura de Lista Duplamente Encadeada com disciplina de PILHA.

3. Uma estrutura de Lista Estática Linear (PILHA) é semelhante a uma pilha de pratos, pois cada prato a ser inserido na pilha será sobre o último elemento, ou seja, no topo; e toda exclusão é executada a partir do topo também.
Então, suponha uma estrutura de Lista Estática Linear (FILA) que possua cinco posições para elementos inteiros.
Execute a sequência de comandos do algoritmo e responda:
1º) InicializaPilha().
2º) VerificaPilhaVazia().
3º) VerificaPilhaCheia().
4º) Insere(21).
5º) Insere(41).
6º) Insere(101).
7º) Insere(90).
8º) VerificaPilhaCheia().

Quais são os valores de cada posição, iniciando pelo índice 0, respeitando a sequência do código acima?
a) 21-41-101-90.
b) 90-101-41-21.
c) 21-101-41-90.
d) 101-90-41-21.
e) 21-90-101-41.

4.



A imagem refere-se à operação de:
a) Inclusão de fila à esquerda.
b) Inclusão de fila à direita.
c) Inclusão de fila acima.
d) Inclusão de fila abaixo.
e) Exclusão de fila acima.

5. P1 é uma pilha com cinco posições, v(1) a v(5), na qual v(5) é o topo. De v(1) até v(5), a pilha P1 está preenchida, respectivamente, com os símbolos Q5, Q3, Q1, Q4, Q2. Há ainda mais duas pilhas, inicialmente vazias, P2 e P3, com o mesmo tamanho.
Fonte: Tecnologia da Informação Algoritmos e Estrutura de Dados Estruturas de dados Pilhas ANO: 2014 BANCA: CESGRANRIO ÓRGÃO: PETROBRAS PROVA: TÉCNICO - TÉCNICO DE INFORMÁTICA (MODIFICADA)

Cheia
P1
Q2
Q4
Q1
Q3
Q5

Qual é a quantidade mínima de movimentos entre as três pilhas para que a pilha P1, originalmente cheia, esteja preenchida de v(5) até v(1), respectivamente, com os símbolos Q1, Q2, Q3, Q4, Q5?

Vazia	Vazia	Vazia
P1	P2	P3

a) 7.
b) 8.
c) 9.
d) 10.
e) 11.

Referências

BATTISTI. **Linguagem C – Alocação dinâmica**. Disponível em: <https://juliobattisti.com.br/tutoriais/katiaduarte/cbasico009.asp>. Acesso em: 13 ago. 2017.

BERTOL, O. F. Disponível em: <http://www.pb.utfpr.edu.br/omero/C/Exercicios/E/PILHAAPO.Htm>. Acesso em: 13 ago. 2017.

IME. **Busca em vetor ordenado**. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/bubi2.html>. Acesso em: 2 ago. 2017.

IME2. **Ponteiros, ponteiros e vetores e alocação dinâmica de memória**. Disponível em: <https://www.ime.usp.br/~mms/mac1222s2013/8%20-%20Ponteiros,%20%20ponteiros%20e%20vetores%20e%20alocacao%20dinamica%20de%20memoria.pdf>. Acesso em: 13 ago. 2017.

LAUREANO, M. **Estrutura de Dados com Algoritmos e C**. Curitiba: Brasport, 2008.

SEDGEWICK, **R. Algorithms in C**. 3. ed. Addison Wesley Longman, 1998.

TENENBAUM, M. A; LANGSAM; Y. AUGENSTEIN, M. J. **Estrutura de Dados Usando C**. São Paulo: Pearson, 1995.

UFRJ. **Estrutura de Dados e Algoritmos**. Disponível em: <http://www.cos.ufrj.br/~rfarias/cos121/filas.html>. Acesso em: 13 ago. 2017.

UFSC. **Curso C**. Disponível em: <http://mtm.ufsc.br/~azeredo/cursoC/aulas/ca60.html>. Acesso em: 13 ago. 2017.

UNICAMP. **Alocação Dinâmica**. Disponível em: <http://www.ic.unicamp.br/~norton/disciplinas/mc1022s2005/03_11.html>. Acesso em: 13 ago. 2017.

UNICAP. **Algoritmos e Estrutura de Dados II**. Disponível em: <https://marciobueno.com/arquivos/ensino/ed2/ED2_11_Pesquisa.pdf>. Acesso em: 2 ago. 2017.

Árvores e grafos

Gisele Alves Santana

Objetivos de aprendizagem

Nesta unidade, você será levado a aprender sobre dois tipos de estruturas de dados muito utilizados na área da computação. Para tanto, os objetivos desta unidade são:

- Compreender o que são grafos e suas aplicações;
- Aprender sobre árvores e suas terminologias;
- Compreender as operações de inserção e exclusão de nós em uma árvore;
- Conhecer os tipos de percursos em uma árvore;
- Aprender as formas de implementações de uma árvore.

Seção 1 | Introdução a grafos e árvores

Nesta seção, você estudará sobre os principais conceitos relacionados aos grafos e às árvores, assim como algumas aplicações desses tipos de estruturas de dados. Esta seção também apresenta as principais terminologias relacionadas a uma árvore e os tipos mais utilizados para a resolução de problemas na área computacional.

Seção 2 | Árvore binária de busca

Nesta seção, você vai aprender sobre o tipo mais utilizado de árvore, que é a árvore binária de busca; exemplos de códigos para a implementação estática e dinâmica; simulações das operações mais importantes para a manipulação dessas árvores e, por fim, os tipos principais de percursos em uma árvore binária de busca, exemplificando graficamente cada um deles.

Introdução à unidade

Os grafos representam um tipo de estrutura de dados muito comum nas aplicações computacionais, especialmente na implementação de jogos. Nesta unidade, serão apresentados alguns conceitos importantes sobre esse tipo de estrutura, exemplificando matematicamente a solução de alguns conceitos relacionados aos grafos, como o cálculo do número de vértices e arcos de um grafo.

Outro tipo muito comum de estrutura de dados aplicado na computação são as árvores, especialmente as árvores binárias de busca. Uma árvore pode ser considerada binária se todos os nós à esquerda do nó raiz forem menores que o nó raiz, assim como se todos os nós à direita do nó raiz forem maiores que o mesmo.

Nesta unidade, serão apresentadas duas maneiras de implementação de uma árvore binária de busca, porém, a implementação dinâmica é mais utilizada. Também serão apresentados exemplos e simulações envolvendo as operações mais importantes relacionadas a esse tipo de árvore, assim como trechos de código na Linguagem C que implementam essas operações para a manipulação dessas estruturas de dados.

Para finalizar, será definido o conceito de percurso ou travessia de uma árvore binária de busca, apresentando e ilustrando quatro tipos de percursos. Funções recursivas que implementam os percursos de árvores binárias também serão exemplificadas.

Seção 1

Introdução a grafos e árvores

Introdução à seção

Esta seção apresenta e ilustra os principais conceitos relacionados aos grafos e às árvores. Os grafos são muito utilizados em aplicações computacionais, especialmente na implementação de jogos. Já as árvores são utilizadas para a organização de dados, principalmente para proporcionar agilidade e rapidez na busca de uma informação.

4.1 Grafos

Como já foi visto na Unidade 1, as estruturas de dados auxiliam na organização das informações, de modo a serem registradas e processadas pelo computador. Alguns exemplos de estruturas de dados são:

- Listas lineares.
- Vetores.
- Árvores.
- Grafos.
- Etc.

Um grafo, ou também chamado de dígrafo, é um conjunto de vértices e arestas (arcos) que interligam pares de vértices distintos TENEMBAUM; LANGSAM; AUGENSTEIN, 2004). Cada aresta de um grafo é um par ordenado de vértices. O primeiro vértice é a ponta inicial da aresta e o segundo é a ponta final. Uma aresta com a ponta inicial "a" e a ponta final "b" é denotado por: a-b, que diz que o arco a-b sai de a e entra em b.

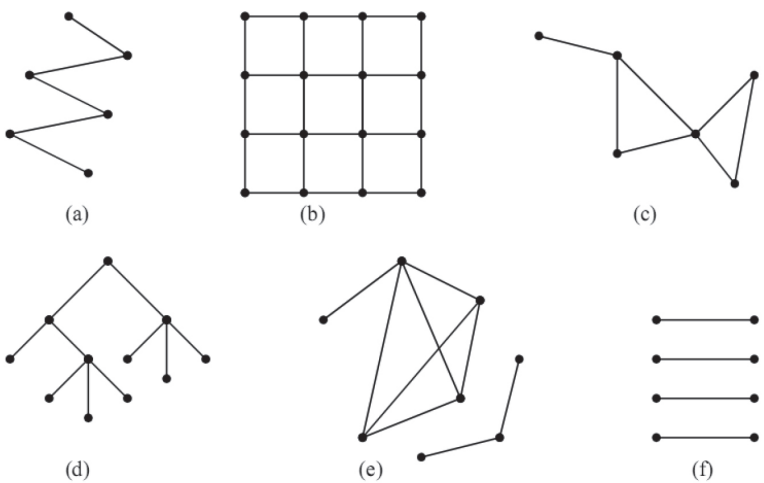
Diversos tipos de aplicações necessitam das estruturas dos grafos. Por exemplo: quando se tem a necessidade de saber se existe um caminho para ir de um objeto a outro, ou calcular a menor distância entre os objetos, ou até mesmo calcular quantos objetos podem ser alcançados a partir de outro objeto.

As árvores são consideradas como subconjunto dos grafos, pois nessas estruturas existe um único caminho que leva a qualquer nó, ou seja, não há possibilidade de se voltar a um nó já visitado a partir de seus filhos (não possui ciclos).

Os grafos são ferramentas muito utilizadas em jogos. Essas estruturas podem ser usadas, por exemplo, para permitir que um personagem caminhe de um ponto a outro de modo eficiente, ou para decidir a próxima estratégia em um jogo, ou até mesmo para resolver um puzzle. Os grafos são comumente aplicados para representar o conjunto de caminhos que um personagem pode navegar no ambiente de um jogo.

Na Figura 4.1, são apresentados vários exemplos de grafos. Os grafos podem ser conexos (a, b, c e d) ou não conexos (e, f). Um grafo é dito conexo quando se pode traçar um caminho que parte de qualquer nó e chega a qualquer outro (TENENBAUM et al., 2004). Um grafo é dito completo quando há uma aresta entre cada par de seus vértices. Se as arestas do grafo são orientadas, o grafo é chamado de orientado.

Figura 4.1 | Exemplos de Grafos



Fonte: adaptada de Tenembaum, Langsam e Augenstein (2004).

Para saber mais

Neste vídeo, é ilustrada a implementação do algoritmo de Dijkstra, que representa outro tipo de método de busca em grafos. Disponível em: <https://www.youtube.com/watch?v=mdWI0WM4EDU>. Acesso em: 16 nov. 2016.

4.1.1 Notação Formal

Um grafo G pode ser formalmente definido como um conjunto de nós ou vértices V interligados por um conjunto de arestas A. Pode-se escrever formalmente da seguinte maneira:

$$G = \{V, A\}$$

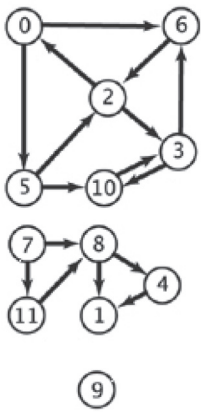
"Muitos grafos possuem pesos associados às arestas. Esse peso pode representar o custo necessário para se mover de um ponto a outro em um grafo. Esse custo pode ser dado em função da distância entre os vértices ou pela dificuldade de locomoção" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 684).

4.1.2 Arcos

Para especificar um grafo, geralmente exibe-se o conjunto de seus arcos. Por exemplo, o conjunto de arcos a seguir define um gráfico com o conjunto de vértices de 0 até 11: 0-5 0-6 2-0 2-3 3-6 3-10 4-1 5-2 5-10 6-2 7-8 7-11 8-1 8-4 10-3 11-8.

A ilustração desse grafo pode ser observada na Figura 4.2.

Figura 4.2 | Grafo e seus arcos



Fonte: adaptada de Tenembaum, Langsam e Augenstein (2004)

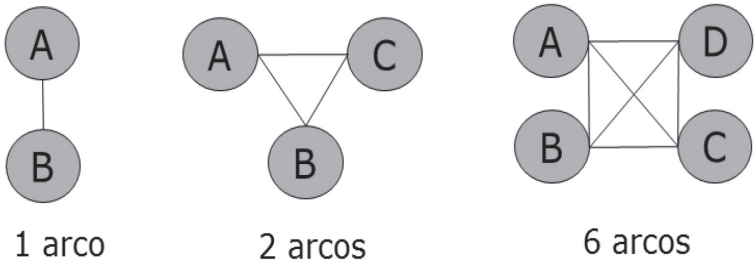
O número de arcos de um grafo é dado pela equação:

$$\frac{N*(N-1)}{2}$$

Onde: N-1 representa todos os vértices, excluindo ele mesmo, e a divisão por 2 significa duas arestas iguais (ida e volta).

Na Figura 4.3 são apresentados exemplos de grafos com 1, 2 e 6 arcos.

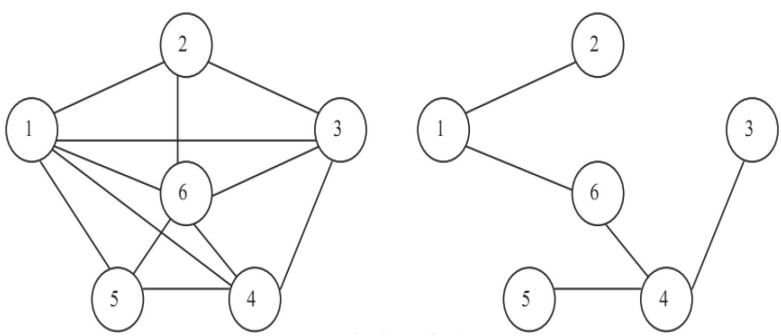
Figura 4.3 | Arcos e vértices



Fonte: elaborada pela autora.

"Um grafo será chamado de completo se todo par ordenado de vértices distintos for um arco. Quando um grafo tem muitos arcos em relação ao seu número, ele é chamado de denso" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 698). Por outro lado, se o grafo possui poucos arcos, é chamado de esparso. A razão entre os vértices e os nós caracteriza se o grafo é denso ou esparso. Grafos esparsos têm poucas conexões por nó e grafos densos possuem muitas. Na Figura 4.4 são apresentados exemplos de grafos denso e esparso.

Figura 4.4 | Exemplo de grafo denso e esparso

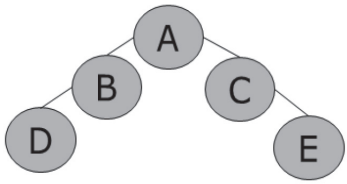


Fonte: adaptada de Tenembaum, Langsam e Augenstein (2004).

4.1.3 Tipos de Grafos

"Existem, basicamente, dois tipos de grafos: grafo não direcional e grafo direcional" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 702). Nos grafos não direcionais, as arestas não são direcionadas ou ordenadas, ou seja, a aresta "V1, V2" é a mesma aresta "V2, V1", conforme ilustrado na Figura 4.5.

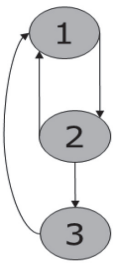
Figura 4.5 | Grafo não direcional



Fonte: elaborada pela autora.

Os grafos direcionais são também chamados de dígrafos. Nesses grafos, as arestas são direcionadas ou ordenadas, ou seja, a aresta "V1, V2" é diferente da aresta "V2, V1". A Figura 4.6 apresenta um exemplo de grafo direcional.

Figura 4.6 | Grafo direcional



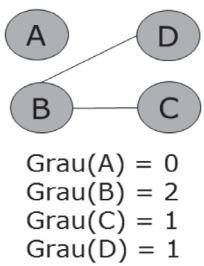
Fonte: elaborada pela autora.

4.1.4 Grau de um Vértice

"O grau é o número de arcos que incidem sobre um vértice. Nos grafos não direcionados, o grau corresponde ao número de arcos que incidem sobre o vértice" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 705).

Já nos grafos direcionados, o grau é o número de arestas que saem dele mais o número de arestas que incidem sobre ele. Um vértice é dito isolado quando seu grau é zero. Na Figura 4.7, o grau do vértice A é igual a zero, pois não existem arestas saindo ou entrando nele.

Figura 4.7 | Graus de um vértice



Fonte: elaborada pela autora.

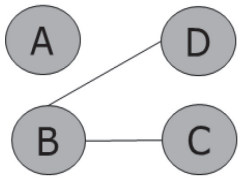
4.1.5 Ciclo

"E um grafo não direcionado, um caminho (v0, v1, ..., v_n) forma um ciclo se *v0* = v_n e o caminho contém, pelo menos, três arestas" TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 700).

Em um grafo direcionado, um caminho (v0, v1, ..., v_n) forma um ciclo se *v0* = v_n e o caminho contém, pelo menos, uma aresta. Os grafos que não possuem ciclos são chamados de acíclicos, já os grafos que possuem ciclos são chamados de cíclicos (MIZRAHI, 2006, p. 121).

O self-loop é um ciclo de tamanho igual a 1. Na Figura 4.8, analisando o ciclo "B C D", percebe-se que os caminhos "B C D", "C D B" e "D B C" formam o mesmo ciclo.

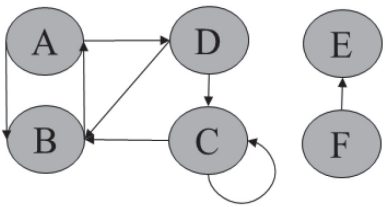
Figura 4.8 | Ciclos de um grafo



Fonte: elaborada pela autora.

Analisando o ciclo "A D C B A" da Figura 4.9, percebe-se que existe um Self-loop no vértice "C", sendo que os caminhos "A D B A", "D B A D" e "B A D B" formam o mesmo ciclo.

Figura 4.9 | Self-loop



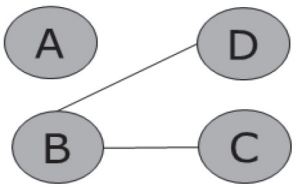
Fonte: elaborada pela autora.

4.1.6 Componentes Conectados

Um grafo não direcionado é conectado quando cada par de vértices está conectado por um caminho. Os componentes conectados são as porções conectadas de um grafo. Um grafo não direcionado é

conectado se ele tem exatamente um componente conectado. Na Figura 4.10, o grafo não é conectado, pois não é possível alcançar o vértice A a partir dos vértices B, C ou D. O grafo {C D B} é um componente conectado do grafo. Inserindo-se o arco {A B}, o grafo passa a ser conectado.

Figura 4.10 | Componentes conectados

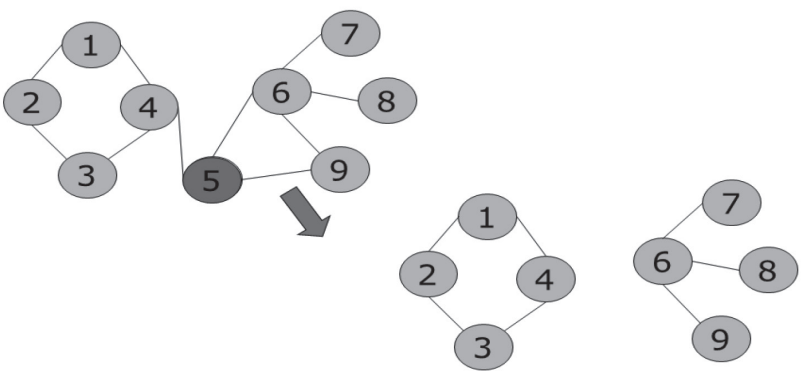


Fonte: elaborada pela autora.

4.1.7 Pontos de Articulação

Os pontos de articulação são vértices que, se forem removidos do grafo, produzirão pelo menos dois componentes conectados. Na Figura 4.11, se o vértice "5" for retirado do grafo, produzirá dois componentes conectados: (1 2 4 3) e (6 7 8 9).

Figura 4.11 | Pontos de articulação



Fonte: elaborada pela autora.

Quando os grafos não possuem nenhum ponto de articulação, são chamados de grafos biconectados.

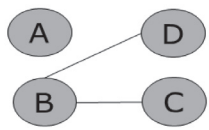
4.1.8 Caminho e Comprimento

"Um caminho de um vértice a para um vértice b em um grafo G = (V;E) é uma sequência de vértices" (v_0, v_1, v_2, ..., v_n) tal que: a = a_0 e b = b_n (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 701).

O comprimento de um caminho é o número de arestas percorridas por esse caminho. Se existir um caminho *c* de a para *b*, então b é alcançável a partir de a via c. Um caminho é simples se todos os vértices do caminho forem distintos. A origem de um caminho é o primeiro vértice, já o término é o seu último vértice. Um caminho é fechado se sua origem coincide com seu término e seu comprimento é maior que 1.

Na Figura 4.12, tem-se um caminho simples. O caminho (C B D) tem comprimento igual a 2 e a aresta *D* é alcançável a partir de C. Já a aresta A não é alcançável a partir de nenhum vértice.

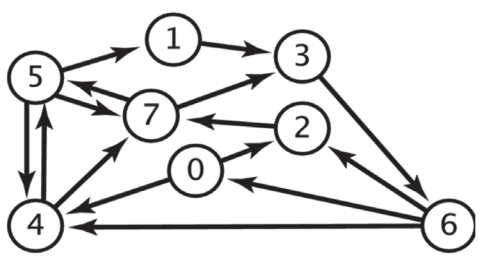
Figura 4.12 | Caminhos de um grafo



Fonte: elaborada pela autora.

O comprimento de um caminho é o número de termos da sequência de vértices menos um. O comprimento de um caminho como "4-7-5-7-5-7", por exemplo, é igual a 5. Se o caminho é simples, seu comprimento é igual ao seu número de arcos. Observe o grafo apresentado na Figura 4.13, que possui diversos caminhos, por exemplo: 0-2-7-3-6; 1-3-6-2-7-3-6-4; 2-7-5-4-7-3 etc.

Figura 4.13 | Caminhos e comprimento



Fonte: adaptada de Tanembaum, Langsam e Augenstein (2004).

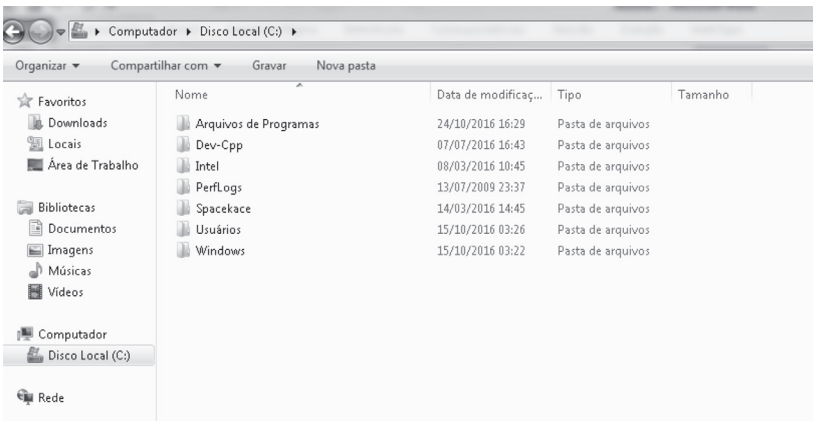
Para saber mais

A apostila a seguir ilustra as operações mais utilizadas para a manipulação de grafos e implementação de algoritmos de busca. Disponível em: <http://www.dainf.ct.utfpr.edu.br/~kaestner/MatematicaDiscreta/Conteudo/Algoritmos/l13-graph-search.pdf>. Acesso em: 19 set. 2017.

4.2 Árvores

"Uma árvore é um tipo de estrutura de dados no qual os dados ficam dispostos de maneira hierárquica. Pode-se dizer que árvores são grafos nos quais existe apenas uma origem e não se pode formar ciclos" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 303). Existem vários tipos de árvores, definidos a partir da quantidade de "filhos" que um elemento ou nó pode ter e de como os elementos são arranjados dentro da árvore. Na computação, as árvores são utilizadas em várias situações, como: estruturas de diretórios em SO, índices para arquivos em disco, estrutura de um arquivo HTML, árvore de decisão em jogos etc. Na Figura 4.14 pode ser observada a estrutura do diretório do Disco Local C.

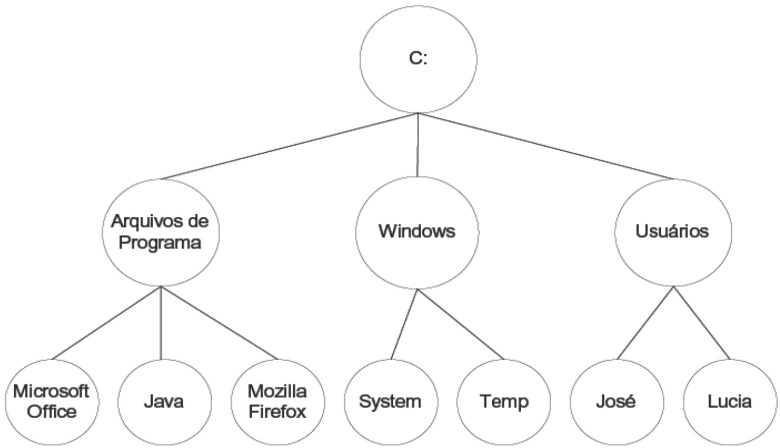
Figura 4.14 | Estrutura dos arquivos do diretório C



Fonte: elaborada pela autora.

A estrutura dos arquivos do diretório C pode ser representada, internamente, por meio de uma árvore, conforme ilustrado na Figura 4.15.

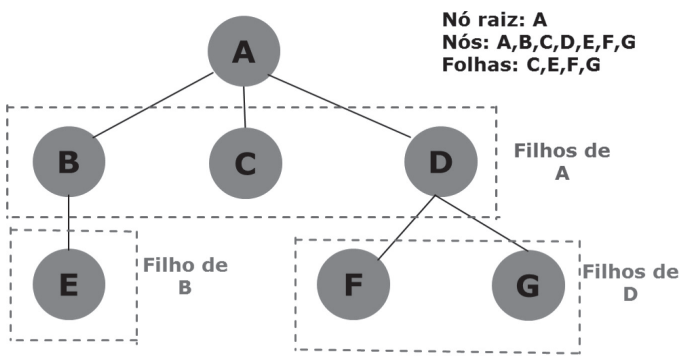
Figura 4.15 | Representação da estrutura dos arquivos do diretório C



Fonte: elaborada pela autora.

Uma árvore é formada por um elemento principal, denominado raiz. A raiz possui ligações com outros elementos, chamados de filhos ou ramos. "Os ramos são ligados a outros elementos que, por sua vez, também possuem outros ramos" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 304). O elemento de uma árvore que não possui ramos é conhecido como nó, folha ou nó terminal. As árvores possuem a tendência de crescer para baixo: a raiz fica no ar enquanto as folhas se enterram no chão. Na Figura 4.16 é apresentado um exemplo de árvore e seus respectivos elementos.

Figura 4.16 | Árvore e seus elementos



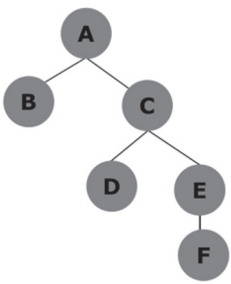
Fonte: elaborada pela autora.

A seguir serão apresentadas as terminologias utilizadas para designar os elementos de uma árvore:

- Subárvore: cada nó da árvore é a raiz de uma subárvore.
- Grau: representa o número de subárvores de um nó.
- Folha: é o nó de grau igual a zero, ou seja, o nó que não possui filhos.
- Nível: a raiz da árvore tem nível 0 (zero) e o nível de qualquer outro nó na árvore é um nível a mais que o nível de seu pai.
- Altura (profundidade): é definida como sendo o nível mais alto da árvore.

Na Figura 4.17 são exemplificadas algumas terminologias de uma árvore que possui altura igual a 3, correspondendo ao seu nível mais alto.

Figura 4.17 | Terminologias de uma árvore



Nó	Grau	Nível
A	2	0
B	0	1
C	2	1
D	0	2
E	1	2
F	0	3

Fonte: elaborada pela autora.

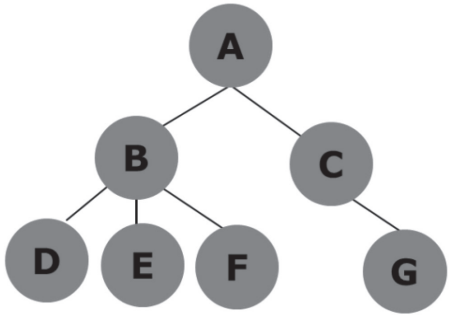
Questão para reflexão

Você consegue citar as vantagens da implementação de árvores para a organização de dados?

4.2.1 Formas de Representação Gráfica

Existem diversas formas para a representação gráfica de uma árvore. A maneira mais utilizada é a representação por meio de grafos, ilustrada na Figura 4.18.

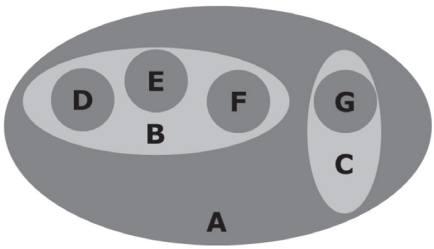
Figura 4.18 | Representação por grafos



Fonte: elaborada pela autora.

Outra maneira de se representar uma árvore é por meio de um diagrama de Venn, conforme ilustrado na Figura 4.19.

Figura 4.19 | Representação por diagrama de Venn



Fonte: elaborada pela autora.

Uma árvore também pode ser representada por meio de parênteses aninhados, de acordo com a Figura 4.20.

Figura 4.20 | Representação por parênteses aninhados

(A (B(D, E, F), C(G)))

Fonte: elaborada pela autora.

4.3 Árvore Binária

Na estrutura de dados, existem diversos tipos de árvores, por exemplo: árvore rubro-negra, AVLS, binária etc. Nesta seção, serão estudadas as árvores binárias, "em que cada nó pode ter no máximo duas subárvores"

(TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 303). Dessa forma, o grau de cada nó pode ser 0, 1 ou 2. Em relação às denominações utilizadas para as subárvores de uma árvore binária, pode-se citar:

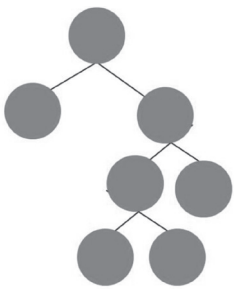
- Subárvore esquerda (E);
- Subárvore direita (D).

As árvores binárias se subdividem em alguns tipos, cujas características serão apresentadas a seguir.

4.3.1 Árvore Estritamente Binária

Neste tipo de árvore binária, cada nó tem 0 (zero) ou 2 subárvores, ou seja, nenhum nó tem "filho único". A Figura 4.21 ilustra uma árvore estritamente binária.

Figura 4.21 | Árvore estritamente binária

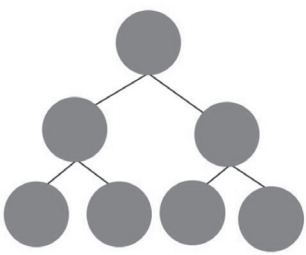


Fonte: elaborada pela autora.

4.3.2. Árvore Binária Cheia

Nas árvores binárias cheias, todos os nós, exceto os nós do último nível, têm exatamente duas subárvores, conforme ilustrado na Figura 4.22.

Figura 4.22 | Árvore estritamente binária



Fonte: elaborada pela autora.

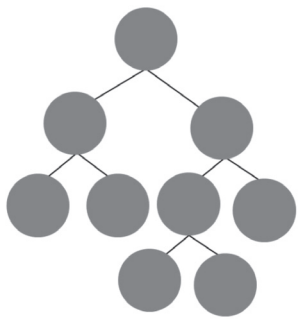
4.3.3 Árvore Binária Balanceada (AVL)

Uma árvore binária é considerada balanceada quando, para cada nó, as alturas de suas subárvores esquerda e direita diferem de, no máximo, uma unidade. Essa diferença é chamada de fator de balanceamento. Dessa maneira, cada nó de uma árvore balanceada pode ter fator de balanceamento entre -1 e +1. Idealmente, uma árvore binária é perfeitamente balanceada quando todos os seus nós têm fatores de balanceamento nulos.

4.3.4 Árvore Binária Completa

A árvore binária completa é um tipo de árvore com grau 0 ou 2, na qual seus nós folhas podem estar apenas no último e no penúltimo nível, conforme observado na Figura 4.23.

Figura 4.23 | Árvore binária completa



Fonte: elaborada pela autora.

Finalizando a seção

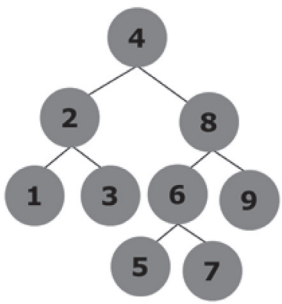
Nesta seção, você aprendeu sobre os grafos, que representam um tipo de estrutura de dados muito comum nas aplicações computacionais; alguns conceitos importantes sobre esse tipo de estrutura, exemplificando a solução de algumas definições relacionadas aos grafos; conceitos básicos, terminologias de uma árvore e os principais tipos delas aplicados na resolução de alguns problemas específicos.

Atividades de aprendizagem

1. Você aprendeu os principais conceitos relacionados aos grafos, incluindo a definição de arestas. Dessa maneira, quantas arestas tem um grafo com vértices de graus 5; 2; 2; 2; 2; 1?

A) 2.
B) 9.
C) 4.
D) 7.
E) 5.

2. Na estrutura de dados, existem diversos tipos de árvores, por exemplo: árvore rubro-negra, AVLS, binária etc. Nas árvores binárias, cada nó pode ter, no máximo, duas subárvores. Dessa forma, o grau de cada nó pode ser 0, 1 ou 2. Analise a árvore binária ilustrada a seguir:



Em relação a ela, é correto afirmar que:

A) A raiz da árvore é representada pelo nó 7.
B) Os filhos do nó 3 são os nós 5 e 7.
C) Os pais do nó 8 são os nós 6 e 9.
D) Os nós 1, 2 e 4 são nós folhas.
E) A raiz da árvore é representada pelo nó 4.

Seção 2

Árvore binária de busca

Introdução à seção

Nesta seção, você vai aprender que uma árvore pode ser considerada binária se todos os nós à esquerda do nó raiz forem menores que ele, bem como se todos os nós à direita forem maiores. Serão apresentadas duas maneiras de implementação de uma árvore binária de busca, assim como exemplos e simulações envolvendo as operações mais importantes relacionadas a esse tipo de árvore. Serão exemplificados trechos de código na Linguagem C que implementam algumas operações para a manipulação dessas estruturas de dados e, para finalizar, será definido o conceito de percurso ou travessia de uma árvore binária de busca, exemplificado por meio de funções recursivas.

4.4 Árvore Binária de Busca

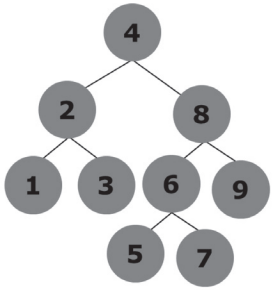
"A árvore binária de busca (ABB) é um tipo de árvore binária, na qual todas as chaves (conteúdo dos nós) da subárvore esquerda são menores que as chaves (conteúdos dos nós) do elemento raiz" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 303). Da mesma forma, todas as chaves da subárvore direita são maiores que a chave do nó raiz.

Os elementos de uma árvore binária de busca são:

- Nós: são todos os itens armazenados em uma árvore.
- Raiz: é o nó do topo da árvore.
- Filhos: são os nós que vêm depois dos outros nós.
- Pais: são os nós que vêm antes dos outros nós.
- Folhas: são os nós que não têm filhos (últimos nós da árvore).

A Figura 4.24 ilustra um exemplo de árvore binária de busca.

Figura 4.24 | Árvore binária de busca



Fonte: elaborada pela autora.

Para a árvore apresentada na Figura 4.24, tem-se:

- **Raiz:** nó 4.
- **Filhos:** o nó 3 é filho do nó 2.
- **Pais:** o nó 8 é pai dos nós 6 e 9.
- **Folhas:** 1, 3, 5, 7 e 9.

Nesta seção, todas as implementações das operações serão baseadas em árvores binárias de busca, uma vez que representam o tipo de árvore mais utilizado e difundido na literatura e em aplicações cotidianas.

4.5 Implementação Estática de uma Árvore Binária de Busca

Na implementação estática de uma árvore binária de busca, os nós são armazenados por nível em um vetor. Assim, se um nó ocupa a posição i na árvore, seus filhos diretos estarão nas posições:

- **2i+1:** nós à esquerda.
- **2i+2:** nós à direita.

Como vantagem da utilização desse tipo de implementação, pode-se citar a economia de espaço de memória, já que o espaço reservado é somente para o armazenamento do conteúdo de cada nó. Quanto à desvantagem, a maior delas está associada aos espaços vazios que se dão quando a árvore não for completa por níveis, ou sofrer eliminação de nós.

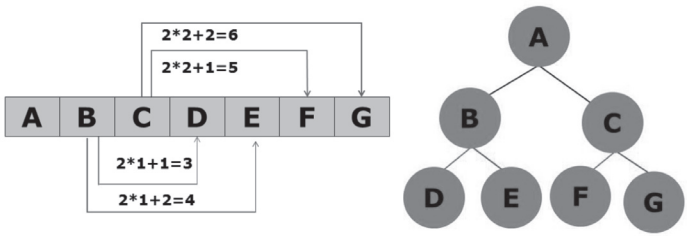
Geralmente, as operações básicas para manipulação de árvores implementadas de forma estática são:

- Criação de uma árvore vazia;

- Definição de um nó;
- Verificação de árvore vazia;
- Impressão dos nós da árvore;
- Exclusão dos elementos da árvore.

Na Figura 4.25, é demonstrado um exemplo de árvore binária de busca e a simulação de implementação estática da mesma.

Figura 4.25 | Implementação estática



Fonte: elaborada pela autora

4.5.1 Definição de um nó

O primeiro passo para a implementação de uma ABB de maneira estática é a definição das informações que serão armazenadas em cada nó da árvore. Outro ponto importante é determinar se um elemento do vetor contém um nó válido ou está vazio. Uma solução é inicializar as posições vazias do vetor com o valor -1. Porém, essa solução funciona apenas no caso de os elementos armazenados nos nós da árvore serem sempre positivos. Outra solução é criar um campo adicional chamado "usado". Assim, cada nó pode ser representado por uma struct, conforme código a seguir:

```
typedef struct {
        int info;
        int usado;
} arvBinaria [MaxElem];
```

4.5.2 Inicialização

Para a inicialização de uma ABB é aconselhável a criação de uma função para inicializar o vetor. A inicialização deve preencher todos os campos usados do vetor como valor *"0"* para que os elementos

possam ser inseridos posteriormente. Deve-se também inicializar o campo *info* com *"0"* para indicar que não há nenhuma informação nesse campo. A seguir, será apresentado um exemplo de função que realiza a inicialização de uma ABB.

```
void inicializa ()
{
   int i;
   for (i = 0; i < MaxElem; i ++)
   {
      arvBinaria [ i ].info = 0;
      arvBinaria [ i ].usado = 0;
   }
}
```

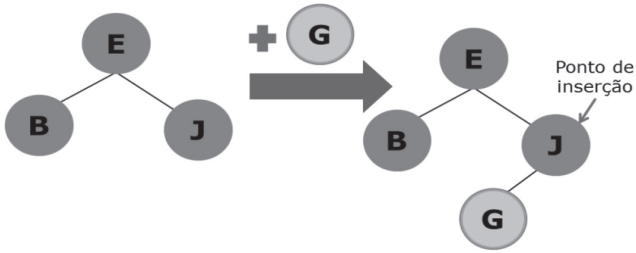
4.5.2 Inserção de Nós

Para a inserção de elementos em uma árvore estática, deve-se pensar que os nós serão armazenados por nível em um vetor. Assim, se um nó ocupa a posição i na árvore, então seus filhos diretos estão nas posições:

- **2i+1:** nós à esquerda
- **2i+2:** nós à direita

O novo nó é sempre inserido como um nó folha. Para facilitar o entendimento da operação de inserção, acompanhe a simulação da inserção de alguns nós em uma árvore binária de busca. Considere a ABB da Figura 4.26, que contém os nós E, B e J, sendo o nó "E" a raiz desta árvore. Suponha que se deseja inserir a letra "G": essa letra é comparada primeiramente com o nó raiz, e como a letra "G" é maior que a letra "E", esse novo nó será inserido na subárvore direita. No nível abaixo, a letra "G" agora é comparada com a letra "J", e como "G" é menor que "J", o novo nó será inserido à esquerda de "J". Como não existem mais nós à esquerda de "J", o novo nó é inserido nesta posição. Lembre-se sempre que os novos nós somente são inseridos como nós folhas.

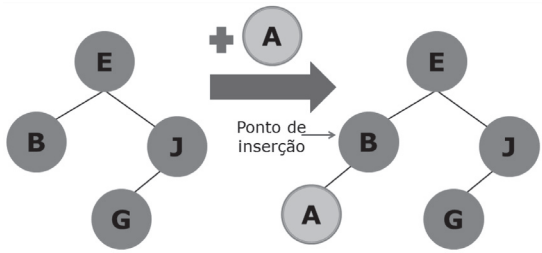
Figura 4.26 | Inserção do nó G



Fonte: elaborada pela autora.

Agora, suponha que se deseja inserir um novo nó que contém a chave "A". Primeiramente, o nó "A" é comparado com a raiz (nó E), e como "A" é menor que "E", o novo nó será inserido na subárvore esquerda. Descendo um nível, o nó "A" é comparado com o nó "B", e como ele é menor que "B", é inserido à esquerda, conforme processo ilustrado pela Figura 4.27.

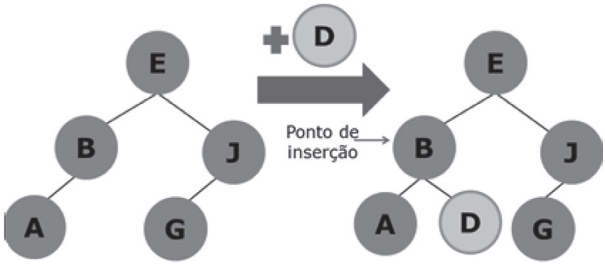
Figura 4.27 | Inserção do nó A



Fonte: elaborada pela autora.

Em outra situação, o nó com chave "D" será inserido nessa árvore. Da mesma maneira, esse nó é comparado com o nó raiz, e como é menor, vai ser inserido na subárvore esquerda. Seguindo o processo de inserção, o novo nó é comparado com o nó "B", e como tem valor maior, é inserido à direita do nó raiz, conforme ilustrado na Figura 4.28.

Figura 4.28 | Inserção do nó D



Fonte: elaborada pela autora.

Todos os outros nós serão inseridos da maneira ilustrada nas figuras anteriores, seguindo os passos:

- Procurar por um local para inserir o novo nó, começando a comparação a partir do nó raiz.
- Para cada nó raiz de uma subárvore, compare: se o novo nó possui um valor menor do que o valor do nó raiz, caminhar para a subárvore esquerda; se o valor é maior que o valor no nó raiz, caminhar para a subárvore direita.
- Se uma referência (filho esquerdo/direito de um nó raiz) nula é atingida (nó folha), inserir o novo nó como sendo filho do nó raiz.

Agora, será exemplificada a inserção de alguns elementos inteiros em uma árvore vazia. Considere a inserção do conjunto de números na sequência: 17, 99, 13, 1, 3, 100. No início, a ABB está vazia, ou seja, não possui nenhum nó. O primeiro nó a ser inserido é o nó com o valor "17". Nesse caso, esse nó é inserido na raiz, conforme observado na Figura 4.29.

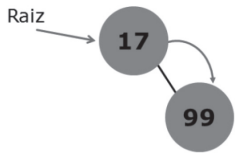
Figura 4.29 | Inserção do nó 17

Fonte: elaborada pela autora.

A inserção do valor "99" inicia-se na raiz, comparando-se esse valor com o valor "17". Como "99" é maior que "17", o novo nó deve ser

inserido na subárvore direita do nó, contendo o valor "17", sendo que esta subárvore está inicialmente nula. Esse processo é ilustrado na Figura 4.30.

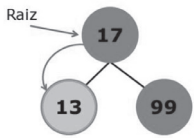
Figura 4.30 | Inserção do nó 99



Fonte: elaborada pela autora.

A inserção do nó com o valor "13" inicia-se na raiz, comparando-se o valor "13" com o valor "17". Como "13" é menor que "17", o novo nó deve ser inserido na subárvore esquerda do nó raiz. Já que o nó 17 não possui descendente esquerdo, o novo nó será inserido na árvore nesta posição, conforme ilustrado na Figura 4.31.

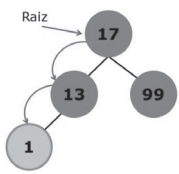
Figura 4.31 | Inserção do nó 13



Fonte: elaborada pela autora.

Para inserir o nó de valor "1", repete-se o mesmo procedimento: compara-se o valor "1" com o valor "17"; como "1" é menor que "17", o novo nó será inserido na subárvore esquerda. Chegando nessa subárvore, encontra-se o nó "13", e como "1" é menor que "13", esse nó será inserido na subárvore esquerda de "13", conforme Figura 4.32.

Figura 4.32 | Inserção do nó 1

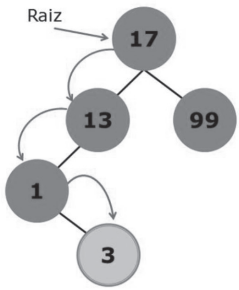


Fonte: elaborada pela autora.

Para inserir o valor 3, deve-se repetir o procedimento:

- Como 3<17, será inserido na subárvore esquerda.
- Chegando na subárvore esquerda, encontra-se o nó 13.
- Como 3<13, desce mais um nível à esquerda.
- Chegando à subárvore esquerda, encontra-se o nó 1, e como 3>1, esse nó será inserido na subárvore direita, conforme ilustrado na Figura 4.33.

Figura 4.33 | Inserção do nó 3

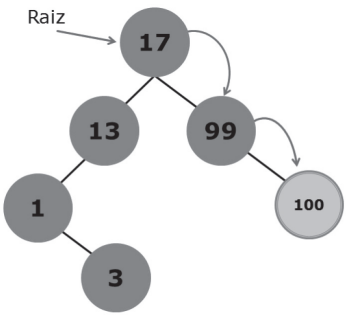


Fonte: elaborada pela autora.

Repete-se o procedimento para realizar a inserção do elemento 100:

- Compara-se o valor do nó a ser inserido com o valor do nó raiz. Como 100 > 17, caminha-se para a subárvore direita.
- Como 100 é maior que 99, caminha-se para a direita e o novo nó é inserido, conforme observado na Figura 4.34.

Figura 4.34 | Inserção do nó 100



Fonte: elaborada pela autora.

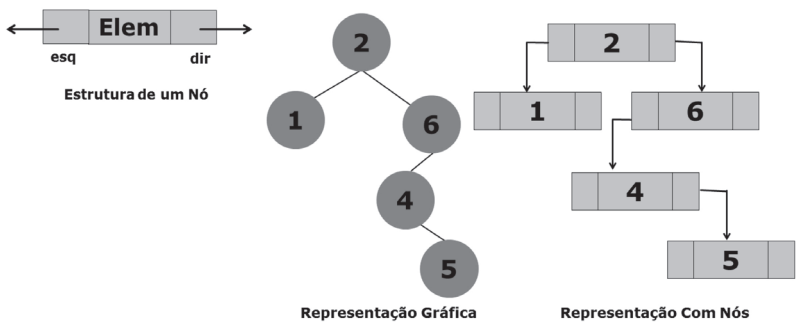
Questão para reflexão

Você consegue imaginar a vantagem principal relacionada à busca de dados em uma árvore binária de busca?

4.5.3 Implementação Dinâmica de uma Árvore Binária de Busca

"A implementação dinâmica é o tipo mais utilizado para manipulação de árvores binárias de busca" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 321). A Figura 4.35 apresenta a estrutura de um nó que possui um campo em que armazena o conteúdo e dois ponteiros: ponteiro da direita e da esquerda. Na figura também pode-se visualizar a representação gráfica de uma árvore, assim como a representação com nós dinâmicos.

Figura 4.35 | Implementação Dinâmica



Fonte: elaborada pela autora.

"A estrutura de dados para uma árvore binária é uma estrutura dinâmica, assim como as listas encadeadas, em que cada nó é representado por um registro," contendo (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 310):

- um campo chave do tipo inteiro, string etc.;
- um ponteiro para as subárvores esquerda e direita;
- outros campos de dados, de acordo com o problema de aplicação.

No código a seguir, é apresentado um exemplo de definição de um nó para uma ABB.

```
struct arv {
   char info;
   struct arv* esq;
   struct arv* dir;
};
```

4.5.3.1 Criação de uma Árvore

Para a criação de uma árvore binária de busca é utilizada uma estrutura em que é alocado espaço para o armazenamento dos nós da árvore, conforme código a seguir:

```
Arv* arv_cria (char c, Arv* e, Arv* d)
{
   Arv* p=(Arv*)malloc(sizeof(Arv));
   p->info = c;
   p->esq = e;
   p->dir = d;
   return p;
}
```

4.5.3.2 Inserção de Nós

A inserção de um novo nó em uma ABB consiste em determinar a posição em que esse nó irá ocupar na árvore, cuja raiz é apontada por um ponteiro.

Se o ponteiro for nulo, então a árvore está vazia e o novo nó se tornará a raiz da árvore, ou seja, o ponteiro passará a apontar para esse novo nó. Segundo Tenembaum, Langsan e Augenstein (2004), algumas regras de inserção para esse caso são:

- O processo de inserção parte do nó raiz (supondo que ele já exista, caso contrário, o novo nó será o elemento a ser inserido na raiz).
- A partir do critério específico de ordenação da árvore, decide-se qual subárvore será percorrida.
- Dentro da nova subárvore repete-se o procedimento, considerando sua raiz, até que se chegue a uma subárvore vazia (sem raiz).

Se o ponteiro do nó raiz não for nulo, três situações podem ocorrer:

- A chave do novo nó é menor que a chave da raiz, logo, o novo nó somente poderá ser inserido na subárvore esquerda da raiz.

Nesse caso, compara-se novamente o novo elemento com a raiz da subárvore esquerda e as mesmas três situações podem ocorrer.

- A chave do novo nó é igual à chave da raiz e, consequentemente, esse novo nó não poderá ser inserido, pois uma árvore binária de busca somente admite uma ocorrência de cada chave.
- A chave do novo nó é maior do que a chave da raiz. Então, o novo nó terá que ser inserido na subárvore direita da raiz. Nesse caso, o valor da chave do novo nó é comparado com a raiz da subárvore direita, sendo possíveis as três situações descritas.

Exceto no caso de ocorrência da situação de chave já existente, o processo é repetido recursivamente até que se encontre uma subárvore vazia (ponteiro nulo). Lembrando que todos os nós serão sempre a raiz de uma subárvore vazia, ou seja, os nós sempre são inseridos como nós folhas.

A seguir, é apresentada uma sugestão de código para a função de inserção de elementos em uma árvore binária de busca.

```
void inserir (struct No **pRaiz, int numero)
{
   if(*pRaiz == NULL){
        * pRaiz = (struct No *) malloc(sizeof(struct No));
        (*pRaiz)→pEsquerda = NULL;
        (*pRaiz)→pDireita = NULL;
        (*pRaiz)→numero = numero;
    }
  else {
        if(numero <(*pRaiz)→numero)
            inserir(&(*pRaiz)→pEsquerda, numero));
        else
            inserir(&(*pRaiz)→pDireita, numero));
  }
 }
```

4.5.3.3 Verificação de uma Árvore Vazia

"A função de verificação indica se uma árvore é ou não vazia, ou seja, compara o valor do ponteiro da raiz da árvore com o valor nulo" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 308). Se o ponteiro tiver o valor igual a nulo, então a árvore está vazia, conforme o código a seguir:

```
int arv_vazia (Arv* a)
{
    return a == NULL;
}
```

4.5.3.4 Liberação de Memória

A função para liberação de memória alocada por uma árvore possui algumas características (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 336):

- As subárvores devem ser liberadas antes de se liberar o nó raiz.
- Retorna uma árvore vazia, representada por NULL.

A seguir, será apresentado um exemplo de função em C que realiza a liberação de memória ocupada por uma ABB.

```
Arv* arv_libera (Arv* a)
{
    if (!arv_vazia(a))
    {
       arv_libera(a->esq); // libera sub-arvore da esq.
       arv_libera(a->dir); // libera sub-arvore da dir.
       free(a); // libera raiz
}
    return NULL;
}
```

A função de impressão dos elementos de uma árvore percorre-a recursivamente, visitando todos os nós e imprimindo seus conteúdos ou chaves, conforme demonstrado no código a seguir.

```
void arv_imprime (Arv* a)
{
    if (!arv_vazia(a))
    {
        printf("%c ", a->info); // mostra raiz
        arv_imprime(a->esq); // mostra sub-arvore esq
        arv_imprime(a->dir); // mostra sub-arvore dir
     }
}
```

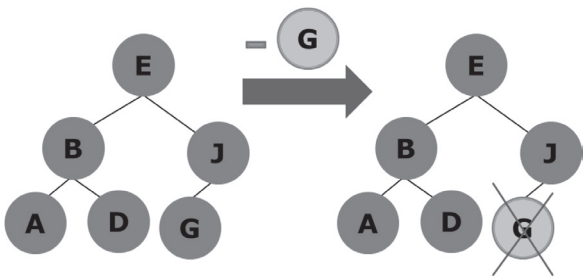
4.5.3.6 Exclusão de Nós

Para excluir um nó de uma árvore binária de busca, deve-se levar em conta três casos distintos: exclusão na folha, exclusão de um nó com 1 filho e exclusão de um nó com 2 filhos. Os três casos serão apresentados a seguir.

a) Exclusão na folha

A exclusão de um nó na folha é o caso mais simples de remoção, pois basta remover o nó folha da árvore. Nesse caso, os ponteiros da esquerda e direita do nó pai são "setados" para NULL. A Figura 4.36 ilustra o processo de exclusão de um nó folha, no caso, o nó "G". Os nós "A" e "D" também podem ser removidos desta forma.

Figura 4.36 | Exclusão de um nó folha

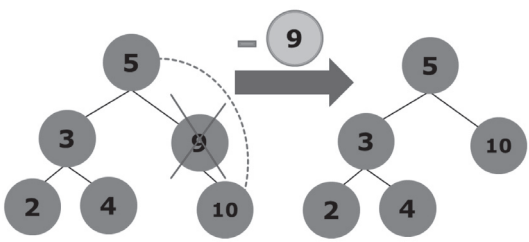


Fonte: elaborada pela autora.

b) Exclusão de um nó com 1 filho

Ao excluir um nó que possui um filho, esse filho assume (sobe) a posição do pai. Nesse caso, o ponteiro apropriado do pai passa a apontar para o filho, conforme ilustrado na Figura 4.37.

Figura 4.37 | Exclusão de um nó com 1 filho



Fonte: elaborada pela autora.

c) Exclusão de um nó com 2 filhos

Nesse caso de exclusão, pode-se proceder de duas maneiras:

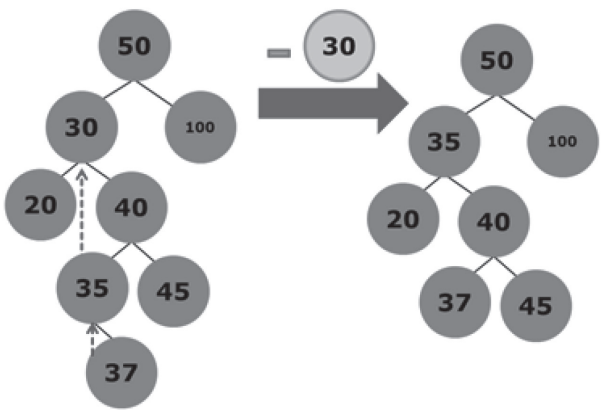
- Substituir o valor do nó a ser retirado pelo valor sucessor (o nó mais à esquerda da subárvore direita).
- Substituir o valor do nó pelo valor antecessor (o nó mais à direita da subárvore esquerda) e, assim, remover-se-á o nó sucessor (ou antecessor).

Os passos para a remoção de um nó com dois filhos podem ser descritos por:

- Encontrar o elemento que precede o elemento a ser retirado na ordenação. Isso equivale a encontrar o elemento mais à direita da subárvore à esquerda;
- Trocar a informação do nó a ser retirado com a informação do nó encontrado;
- Excluir o nó encontrado.

Na Figura 4.38, deseja-se remover o nó de valor "30". Esse nó possui como sucessor imediato o valor "35" (nó mais à esquerda da sua subárvore direita). Excluindo o nó de valor "30", o nó de valor "35" será promovido no lugar do nó a ser excluído, enquanto a sua subárvore direita será promovida para subárvore esquerda do nó com valor "40".

Figura 4.38 | Exclusão de um nó com 2 filhos



Fonte: elaborada pela autora.

A seguir, será apresentada uma função que realiza os três tipos de exclusões de nós apresentados nesta seção.

```
Arv* exclusao (Arv* r, int v) {
   if (r == NULL)
      return NULL;
   else if (r->info > v)
       r->esq = retira(r->esq, v);
   else if (r->info < v)
       r->dir = retira(r->dir, v);
   else { // achou o elemento
      if (r->esq == NULL && r->dir == NULL) { // elemento sem filhos
          free (r);    r = NULL;    }
      else if (r->esq == NULL) { // só tem filho à direita
          Arv* t = r;
          r = r->dir;      free (t);       }
      else if (r->dir == NULL) { // só tem filho à esquerda
          Arv* t = r;
          r = r->esq;      free (t);       }
      else {       // tem os dois filhos
         Arv* pai = r;
         Arv* f = r->esq;
         while (f->dir != NULL) {
             pai = f;
             f = f->dir;    }
         r->info = f->info; // troca as informações
        f->info = v;      r->esq = retira(r->esq,v);
       }
     }
     return r;
}
```

Para saber mais

Este vídeo contém um breve apanhado sobre o funcionamento básico dos algoritmos de inserção e remoção de elementos de uma árvore binária de busca. Disponível em: <https://www.youtube.com/watch?v=XZ0MEDhb4oE>. Acesso em: 19 nov. 2017.

4.6 Percursos

"Percurso é o caminho realizado pelos nós da árvore com o objetivo de consultar ou alterar a informação neles contida" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 510). Existem quatro tipos de percursos:

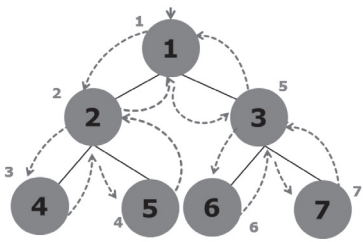
- Percurso Pré-ordem.
- Percurso In-ordem.
- Percurso Pós-ordem.
- Percurso em Nível.

A seguir serão apresentados os principais tipos de percursos.

4.6.1 Percurso Pré-Ordem (R, E, D)

Neste percurso, visita-se primeiramente a raiz e depois as subárvores esquerda e direita, respectivamente. O percurso é iniciado pela raiz da árvore; assim que o nó é visitado, o valor é mostrado (1ª passagem). Dessa maneira, na árvore ilustrada na Figura 4.39 o resultado do percurso é: 1, 2, 4, 5, 3, 6, 7.

Figura 4.39 | Percurso Pré-Ordem



Fonte: elaborada pela autora.

No código a seguir, tem-se uma função recursiva, em que é impresso o conteúdo de cada nó da árvore binária, percorrendo a árvore na ordem: raiz, subárvore esquerda e subárvore direita.

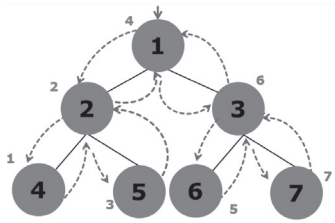
```
void In_Ordem(Arv raiz)
{
    if (raiz == NULL)
        return 0;
   In_Ordem(raiz->esq);
   printf("%d\n", raiz->info);
   In_Ordem(raiz->dir);
}
```

4.6.2 Percurso In-Ordem (E, R, D)

No percurso In-Ordem, percorre-se primeiramente a subárvore

da esquerda, visita-se a raiz e, por último, percorre-se a subárvore da direita. Assim, esse percurso é iniciado pela raiz da árvore, caminha-se inicialmente pelos nós da esquerda, somente exibindo os valores quando todos à esquerda já tiverem sido visitados (2ª passagem). O resultado do percurso in-ordem da árvore ilustrada na Figura 4.40 é: 4, 2, 5, 1, 6, 3, 7.

Figura 4.40 | Percurso In-Ordem



Fonte: elaborada pela autora.

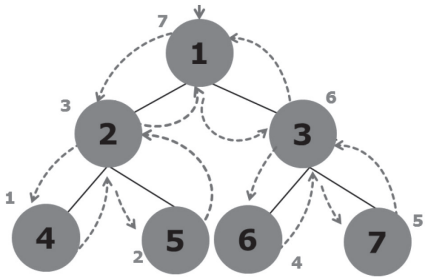
A seguir, será apresentado um exemplo de código que realiza o percurso in-ordem, no qual a árvore é percorrida na seguinte ordem: subárvore esquerda, raiz e subárvore direita.

```
void In_Ordem(Arv raiz)
{
   if (raiz == NULL)
        return 0;
  In_Ordem(raiz->esq);
  printf("%d\n", raiz->info);
  In_Ordem(raiz->dir);
}
```

4.6.3 Percurso Pós-Ordem (E, D, R)

"Neste tipo de percurso, percorre-se a subárvore da esquerda, logo após, percorre-se a subárvore da direita e, finalmente, visita-se a raiz" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 520). Dessa maneira, inicia-se o percurso pela raiz da árvore, caminhando a princípio pelos nós da esquerda e, em seguida, pelos nós da direita, apenas exibindo os valores quando todos os nós descendentes já tiverem sido visitados (3ª passagem). O resultado do percurso pós-ordem da árvore ilustrada na Figura 4.41 é: 4, 5, 2, 6, 7, 3, 1.

Figura 4.41 | Percurso Pós-Ordem



Fonte: elaborada pela autora.

No código a seguir, será apresentada uma função que realiza o percurso pós-ordem, na qual a árvore é percorrida na seguinte ordem: subárvore esquerda, subárvore direita e raiz.

```
void Pos_Ordem(Arv raiz)
{
   if (raiz == NULL)
        return 0;
   Pos_Ordem(raiz->esq);
   Pos_Ordem(raiz->dir);
   printf("%d\n", raiz->info);
}
```

4.6.4 Percurso em Nível

Neste tipo de percurso, a árvore é percorrida no sentido de cima para baixo e da esquerda para direita. "É o percurso mais fácil de ser compreendido, porém, o mais difícil de ser programado, já que é necessário utilizar uma fila em um algoritmo iterativo" (TENEMBAUM; LANGSAM; AUGENSTEIN, 2004, p. 521). A seguir, um exemplo de código que apresenta o percurso em nível em uma árvore binária. A função utiliza uma fila implementada em um vetor fila, em que i é o índice do primeiro item da fila e f-1 é o índice do último elemento, supondo que todos os elementos da fila são diferentes de NULL.

```
void Nivel (Arv raiz) {
   Arv *fila;
   int i, f;
   fila = malloc(count(raiz) * sizeof (Arv));
   fila[0] = raiz;
   i = 0; f = 1;
   while (f > i) {
      raiz = fila[i++];
      printf("%d\n", raiz->info);
      if (raiz->esq != NULL)
            fila[f++] = raiz->esq;
      if (raiz->dir != NULL)
            fila[f++] = raiz->dir;
   }
   free(fila);
}
```

Para saber mais

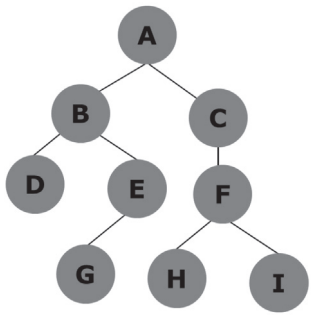
Nesta página, são mostradas as diferentes maneiras de se caminhar em uma árvore por meio de animações de fácil compreensão. Disponível em: <http://www.ufpa.br/sampaio/curso_de_estdados_1/arvores/pagina_10_07_2001/aula26.htm>. Acesso em: 19 set. 2017.

Finalizando a seção

Neste tema, você aprendeu conceitos básicos sobre árvores binárias, assim como alguns códigos responsáveis pela implementação de operações básicas com esse tipo de estrutura de dados. Algumas simulações de inserção e exclusão de nós foram realizadas, já que essas operações são consideradas as mais trabalhosas para implementação. Visto que as árvores podem ser implementadas de forma estática ou dinâmica, foram apresentadas operações básicas para a manipulação de ambas; soluções para quatro tipos de percursos, exemplos de códigos para sua implementação e, por fim, foram também apresentados alguns tipos de percurso em uma árvore binária.

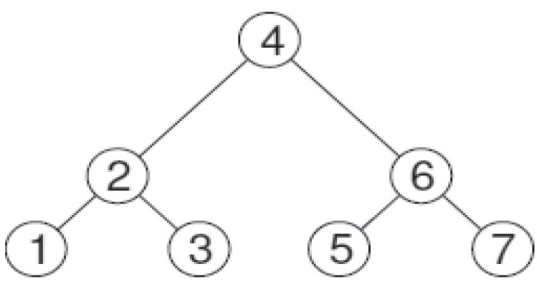
Atividades de aprendizagem

1. Observe a árvore ilustrada na figura a seguir:



Em relação à profundidade, nível e parentesco de cada nó, analise as seguintes afirmativas:
I) A nó E está no nível 2.
II) O nó H está no nível 3.
III) O pai do nó A é o nó B.
IV) A profundidade da árvore é 4.
Assinale a alternativa correta:
a) As afirmativas I, II, III e IV estão corretas.
b) Apenas as afirmativas I e II estão corretas.
c) Apenas as afirmativas I e III estão corretas.
d) Apenas as afirmativas II e IV estão corretas.
e) Apenas as afirmativas III e IV estão corretas.

2. Realize os quatro tipos de percurso na árvore abaixo, escrevendo a sequência dos nós visitados.



Fique ligado

Nesta unidade, você aprendeu sobre os grafos, que representam um tipo de estrutura de dados muito comum nas aplicações computacionais, especialmente na implementação de jogos. Foram apresentados alguns conceitos importantes sobre esse tipo de estrutura, exemplificando matematicamente a solução de alguns conceitos relacionados aos grafos. Exemplos e simulações envolvendo as operações mais importantes relacionadas aos grafos foram apresentados.

Você também aprendeu os conceitos básicos sobre árvores binárias, assim como alguns códigos responsáveis pela implementação de operações básicas com esse tipo de estrutura de dados. Algumas simulações de inserção e exclusão de nós foram realizadas, já que essas operações são consideradas as mais trabalhosas para implementação. Foi visto que as árvores podem ser implementadas de forma estática ou dinâmica; logo, foram apresentadas operações básicas para manipulação de ambas.

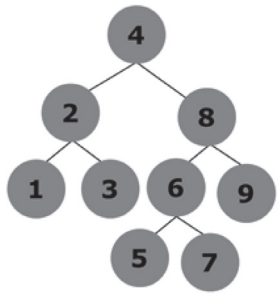
Finalizando, foram apresentados alguns tipos de percursos em uma árvore binária, com algumas soluções para quatro tipos de percursos, assim como exemplos de códigos para sua implementação.

Para concluir o estudo da unidade

Nesta unidade, você conheceu os principais tipos de percursos que podem ser realizados em uma árvore binária de busca. Também existem métodos para busca de dados em grafos, como: busca em profundidade (DFS) e busca em largura (BFS), sendo que a principal diferença entre essas buscas está relacionada à estrutura de dados auxiliar que é empregada. Enquanto a busca BFS utiliza uma fila de vértices, a busca DFS utiliza uma pilha que armazena os vértices de grafo.

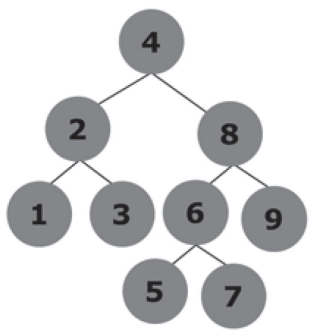
Atividades de aprendizagem da unidade

1. Na árvore binária de busca (ABB), todas as chaves (conteúdo dos nós) da subárvore esquerda são menores que as chaves (conteúdos dos nós) do elemento raiz. Da mesma forma, todas as chaves da subárvore direita são maiores que a chave do nó raiz. Considere a ABB ilustrada na figura a seguir:



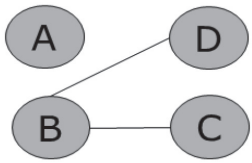
Suponha a inclusão do nó "13".
Em qual posição esse novo elemento será incluído na árvore?
a) Na subárvore direita de 9.
b) Na subárvore esquerda de 9.
c) Na subárvore esquerda de 7.
d) Na subárvore direita de 9.
e) Na subárvore esquerda de 5.

2. Na árvore binária de busca (ABB), todas as chaves (conteúdo dos nós) da subárvore esquerda são menores que as chaves (conteúdos dos nós) do elemento raiz. Da mesma forma, todas as chaves da subárvore direita são maiores que a chave do nó raiz. Considere a ABB ilustrada na figura a seguir:



Suponha a exclusão do nó "3".
O que acontecerá com essa árvore?
a) O nó 1 passa a ser o pai do nó 2.
b) O nó 2 passa a ser a raiz da árvore.
c) O nó 1 é deslocado para a subárvore direita do nó 2.
d) O nó 3 é simplesmente removido.
e) A subárvore esquerda do nó 4 deixa de existir

3. Os grafos representam um tipo de estrutura de dados muito comum nas aplicações computacionais, especialmente na implementação de jogos. Os grafos são compostos por vértices e arestas. O grau de um vértice é o número de arcos que incidem sobre um vértice. Neste contexto, analise o grafo a seguir:



Qual é o grau do vértice C?
a) 1.
b) 0.
c) 2.
d) 3.
e) 4.

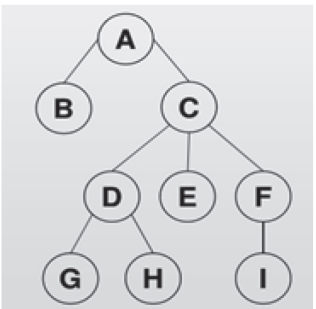
4. A característica principal de uma árvore binária de busca (ABB) é a existência de duas subárvores: esquerda e direita, o que facilita a pesquisa por elementos. Suponha que foram inseridos, nesta sequência, os seguintes elementos em uma ABB: 10, 5, 15, utilizando a seguinte função para a inserção:

```
void inserir (struct No **pRaiz, int numero)
{
   if(*pRaiz == NULL){
        * pRaiz = (struct No *) malloc(sizeof(struct No));
        (*pRaiz)→pEsquerda = NULL;
        (*pRaiz)→pDireita = NULL;
        (*pRaiz)→numero = numero;
    }
  else {
        if(numero <(*pRaiz)→numero)
            inserir(&(*pRaiz)→pEsquerda, numero));
        else
            inserir(&(*pRaiz)→pDireita, numero));
  }
 }
```

Imagine que o próximo elemento a ser inserido é o nó de valor 17. Na execução dessa função para a inserção desse novo elemento, é correto afirmar que:

I) A instrução *if(numero <(*pRaiz)→numero)* somente será falsa quando o valor a ser inserido for maior que o nó sendo comparado.
II) A função inserir, por ser recursiva, chama a si mesma até que o local correto para a inserção do novo nó seja encontrado.
III) Como o valor a ser inserido (valor 17) é maior que o valor do nó raiz, a navegação pela árvore é realizada por meio da subávore da direita do nó raiz.
IV) A instrução if(*pRaiz == NULL) somente será verdadeira se for encontrado um nó folha, ou seja, um local no qual o novo nó será inserido.
Assinale a alternativa correta.
a) apenas a afirmativa I está correta.
b) apenas a afirmativa IV está correta.
c) apenas as afirmativas II e III estão corretas.
d) apenas as afirmativas I e III estão corretas.
e) as afirmativas I, II, III e IV estão corretas.

5. A estrutura não linear de maior aplicação em computação, provavelmente, é a estrutura de árvore ou simplesmente árvore. Analise a árvore ilustrada na figura abaixo:



A respeito dessa árvore, é correto afirmar que:
a) O nó "A" possui nível igual a 0 (zero).
b) O nó "B" possui nível igual a 0 (zero).
c) O nó "E" possui grau igual a 1 (um).
d) O nó "C" possui grau igual a 5 (cinco).
e) A altura da árvore é igual a 7 (sete).

Referências

JUNIOR, Dilermando Piva; et al. **Estrutura de Dados e Técnicas de Programação**. 1. ed. São Paulo: Elsevier - Campus, 2014.

MIZRAHI, Victorine Viviane. **Treinamento em linguagem C++**. São Paulo: Makron, 2006.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: Pearson Makron Books, 2004.

VELOSO, Pauloet al. **Estrutura de dados**. Rio de Janeiro: Campus, 1986.

Anotações

Anotações

Anotações

Anotações

Anotações

Anotações

Anotações

unopar

ISBN 978-85-522-0314-8