ESTRUTURA E CLASSIFICAÇÃO DE DADOS

CARLOS DANILO LUZ

SOBRE OS AUTORES

Carlos Danilo Luz

Graduado em Redes de Computadores pelo Centro de Ensino Superior de Maringá – CESUMAR – 2009.
Especialista em Educação a Distância e Tecnologias Educacionais pelo Centro Universitário de Maringá – UniCesumar – 2016.
Especialista em Gestão Estratégica de Pessoas pelo Centro Universitário de Maringá – UniCesumar – 2017.

Possui graduação em Redes de Computadores pelo Centro de Ensino Superior de Maringá, especialista em Educação a Distância e Tecnologias Educacionais, especialista em Gestão Estratégica de Pessoas, cursando especialização em Docência no Ensino Superior. Mestrando em Ciência da Computação pela Universidade Estadual de Maringá - UEM. Atualmente é professor pela Secretaria de Educação do Estado do Paraná e Tutor Operacional IV - Avaliação do Centro de Ensino Superior de Maringá. Atua principalmente nos seguintes temas: Programação PHP, BACK END, Banco de Dados e Redes de Computadores. Conta com experiência de 9 anos em desenvolvimento de sistemas web e softwares.

Introdução

Caro(a) aluno(a), seja bem-vindo(a) a este livro, que foi elaborado especialmente para que você possa conhecer os conceitos básicos e iniciais de lógica de programação e estrutura de dados. O livro está dividido em quatro unidades, nas quais aprenderemos os conceitos iniciais da linguagem de programação C, que é uma linguagem muito popularizada por ser considerada base para o desenvolvimento das linguagens atuais, como C#, JAVA, PHP, entre outras. Todas as unidades apresentam exemplos utilizando a linguagem C.

Na primeira unidade, veremos um breve histórico da linguagem C, suas características e os conceitos iniciais sobre programação. Estudaremos como é a estrutura básica de um programa em linguagem C, a declaração de variáveis e manipulação de dados através de entrada e saída de dados. Conheceremos também os operadores e expressões, entre outros conceitos que o(a) ajudarão no desenvolvimento de seus programas em linguagem C, através disso, você, aluno(a), terá o conhecimento necessário para desenvolver os seus primeiros programas em linguagem C.

Na segunda unidade, vamos relembrar os conceitos sobre sistemas numéricos, pois vamos utilizá-los ao estudarmos sobre alocação de dados na memória. Será apresentada a estrutura de dados em forma de árvore, que é muito utilizada para a organização dos dados na memória por ser de fácil manipulação pelo sistema. Iremos estudar também sobre as listas lineares, que são um tipo de vetor para armazenarmos e recuperarmos dados. Veremos ainda as listas encadeadas, duplamente encadeadas, circulares e ordenadas. Por fim, será visto o conceito das duas estruturas de dados mais importantes, estrutura de FILA e PILHA, conceitos esses utilizados para diversos algoritmos de busca e ordenação de dados.

Na terceira unidade, aprenderemos os conceitos sobre ordenação de dados, pois quanto mais organizados e ordenados os elementos, mais rápida será a manipulação deles. Serão compreendidos alguns dos algoritmos de ordenação mais comuns como o BubbleSort (método bolha) , InsertionSort, ShellSort (concha), MergeSort (dividir para conquistar) entre outros.

Por fim, na quarta unidade, trabalharemos com busca e classificação de dados do vetor, serão vistos métodos simples de busca de elementos em vetores ordenados ou não ordenados através da busca sequencial, desde as mais complexas até as por interpolação. Para finalizar, também veremos como é possível classificar os elementos dentro de um vetor com a tabela hash.

Desejo a você, aluno(a), uma ótima leitura e bons estudos!

UNIDADE I

Introdução à Linguagem de Programação C

Carlos Danilo Luz

Caro(a) aluno(a)! Seja bem-vindo(a)! Nesta unidade, aprenderemos os conceitos iniciais da linguagem de programação C, que é muito popularizada por ser considerada como base para o desenvolvimento das linguagens atuais, como C#, JAVA, PHP, entre outras. Será apresento a você alguns exemplos de código fonte em C. Estudaremos como é a estrutura básica de um programa em linguagem C, os tipos de dados disponíveis, como declarar variáveis e manipulação de dados através de entrada e saída de dados. Conheceremos também os operadores e expressões. Com base nesse conhecimento, podem-se iniciar os nossos primeiros programas em C.

Conceitos Iniciais sobre Linguagem de programação C

Em nosso cotidiano, utilizamos diversos softwares e aplicativos para nos auxiliar em nossas atividades, sendo eles programa de gestão empresarial conhecidos como ERP (*Enterprise Resource Planning* ou, na tradução literal, Planejamento dos Recursos da Empresa), sistemas bancários, aplicativos de celular, entre diversos outros. Esses softwares ou programas (como alguns os conhecem) são desenvolvidos através de linguagens de programação.

1FIGURA 1.59 - Linguagem de Programação C FONTE: SCANDINAVIANSTOCK, 123RF.

Podemos definir que as linguagens de programação são instruções, sendo elas um conjunto de palavras, compostas por comandos e regras que dão origem ao código fonte do programa. Após a construção do código fonte, ele é interpretado e executado pelo processador do computador ou do celular.

Encontramos hoje à disposição diversas linguagens de programação como JAVA, Python, PHP, Visual Basic, Delphi, C#, entre outras, o que define qual a melhor linguagem a se utilizar é qual aplicação ou programa será efetuado. Muitas dessas linguagens têm suas instruções iguais ou parecidas, através da linguagem de programação C, podemos compreender como são os códigos fontes de um programa de uma forma simples e prática.

O que é a linguagem de programação C

A linguagem de programação C teve seu início na década de 70, e Dennis Ritchie foi o responsável por inventá-la e implementá-la nos laboratórios Bell da companhia AT & T. A linguagem C teve influência direta da linguagem chamada B, inventada por Ken Thompson, destinada inicialmente apenas para sistemas operacionais UNIX até ganhar espaço entre os desenvolvedores e sistemas.

> Com a popularidade dos microcomputadores, um grande número de implementações de C foi criado. Quase por milagre, os códigos-fontes aceitos por essas implementações eram altamente compatíveis. (Isto é, um programa escrito com um deles poderia normalmente ser compilado com sucesso usando-se um outro)
>
> (SHILDT, 1997, p.3).

A linguagem C se utiliza de uma sintaxe muito simples e compacta, parte de sua popularidade se dá também pelas combinações de diferentes funções, além de não estar vinculada a qualquer tipo de sistema ou hardware específico, por isso, pode ser compilada com sucesso por vários sistemas e também ser executada por qualquer computador que tenha suporte à linguagem.

Essa linguagem é classificada como nível médio, mas isso não significa que ela não é poderosa, Shildt (1986, p.1) afirma que "***Uma linguagem de nível médio fornece aos programadores um conjunto mínimo de declarações de controle e manipulação de dados, que eles poderão utilizar para definir construções de alto nível***". A linguagem C serve como apoio para diversas linguagens de alto nível, como PASCAL ou BASIC, podendo também ser facilmente convertida para algum tipo de linguagem de alto nível.

O motivo de compreendermos a C como nossa primeira linguagem de programação é pelo fato de que ela é lida com muita facilidade, uma vez que, estando familiarizado(a) com C, pode-se aplicar a mesma lógica de desenvolvimento para as demais linguagens disponíveis no mercado.

¶ Fique por dentro

Linguagens de Baixo, Médio e Alto Nível

- Linguagens de baixo nível são códigos executados diretamente pelo computador, conhecidas também por linguagem binária, pois é composta por 0 e 1.
- Linguagens de médio nível são utilizadas para escrever programas com base em sua linguagem, podem ser procedimental ou não procedimental.
- Linguagens de alto nível são linguagens que se aproximam da linguagem humana.

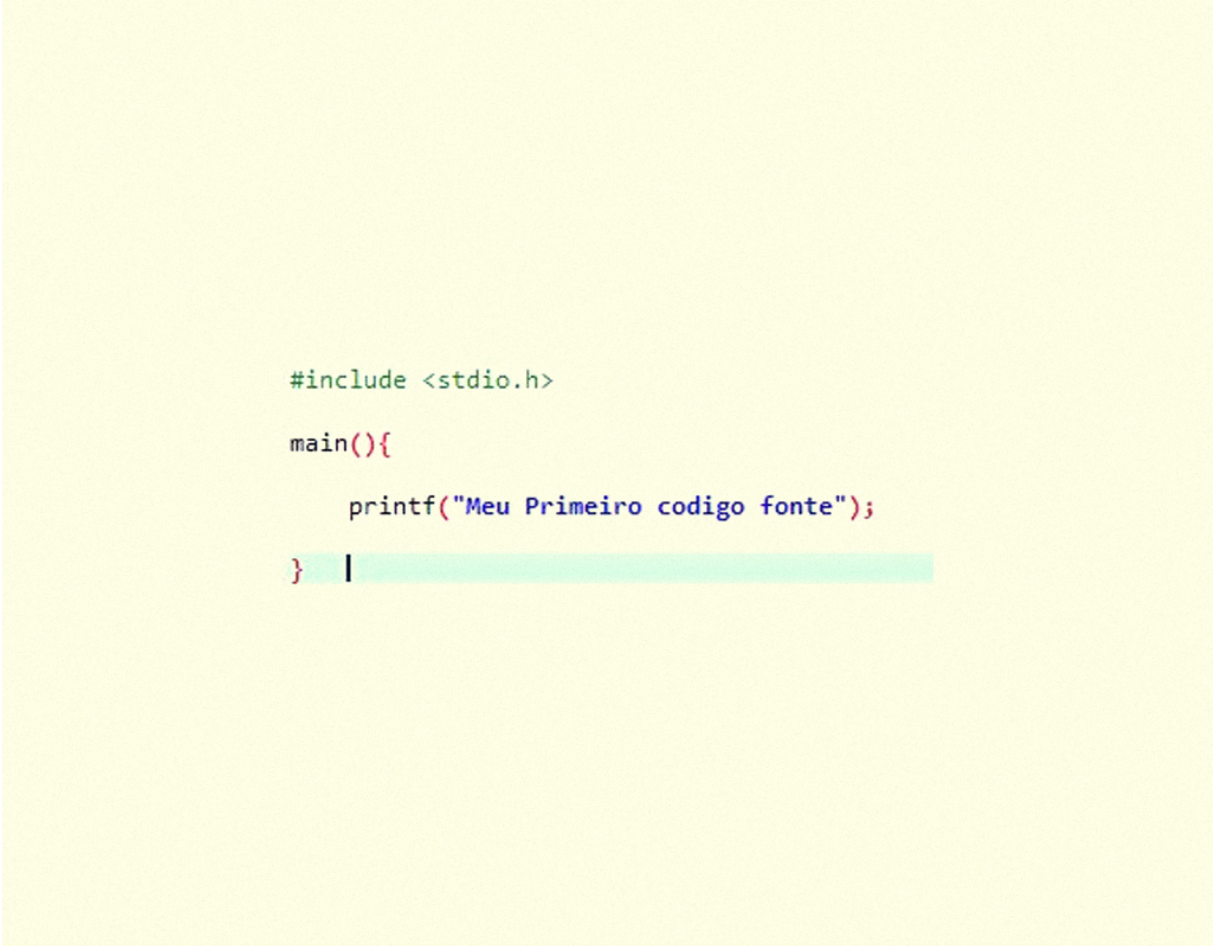
Linguagens de computação em níveis

FONTE adaptado de Schildt (1986).

BAIXO NÍVEL	MÉDIO NÍVEL	ALTO NÍVEL
Macro-assembler Assembler	C C++ FORTH	Ada Modulo-2 Pascal COBOL FORTRAN BASIC

Estrutura do arquivo de linguagem de programação C

O desenvolvimento de um programa em linguagem C se inicia com a escrita do código fonte por meio de um editor e termina na compilação do código, gerando um programa executável, alguns dos principais compiladores são: GCC, Dev C++, C++ Builder, Turbo C e Visual C#. Não se preocupe, pois o código fonte não se altera independentemente do compilador escolhido. Para compreendermos melhor vamos construir nosso primeiro programa em C, como na Figura 1.2.

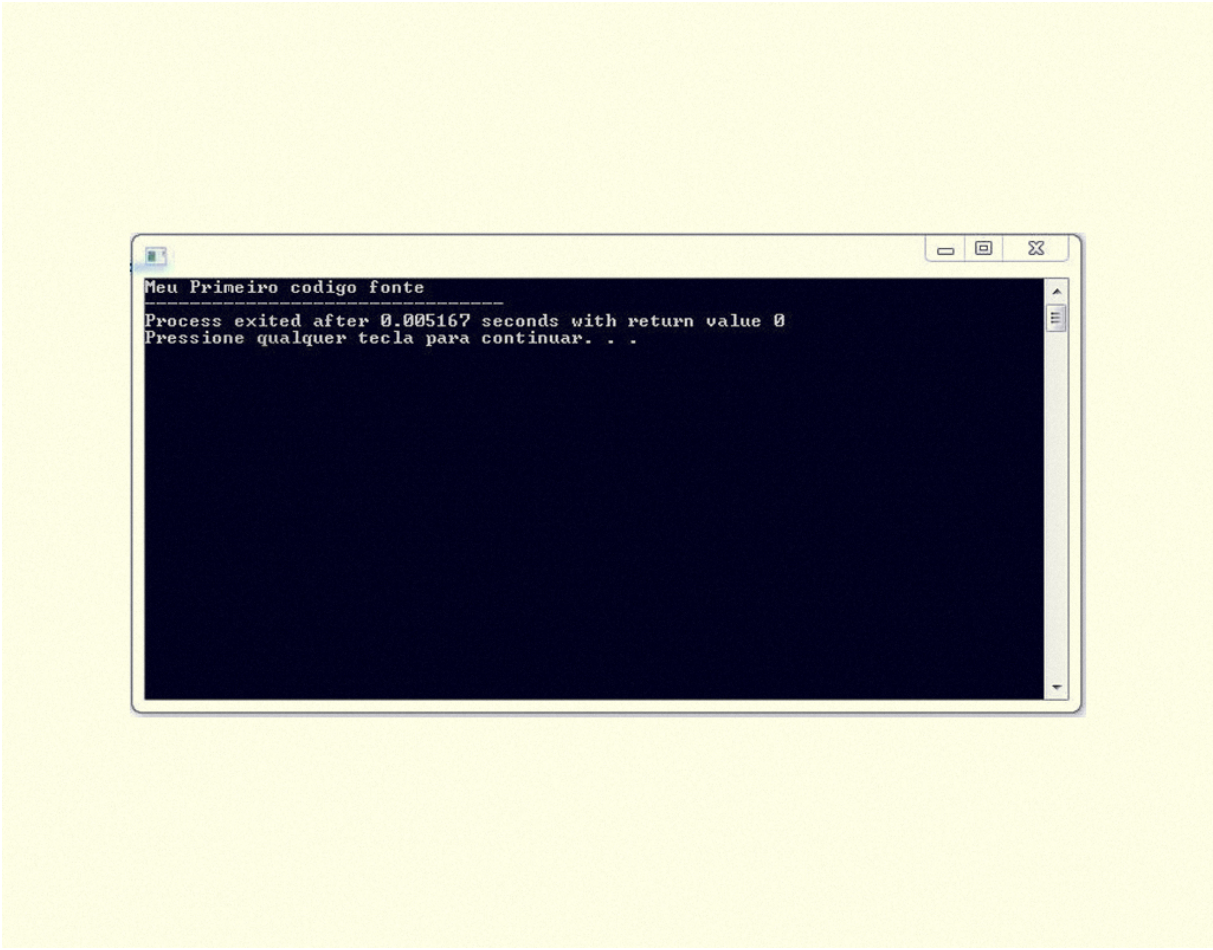


1FIGURA 2.59 - Meu Primeiro programa em C FONTE: Dev-C ++.

Prezado(a) aluno(a), fique tranquilo, pois vamos analisar cada linha de nosso código.

Ele se inicia com a função **#include <stdio.h>**, na qual estamos incluindo a biblioteca **stdio.h,** em que há declarações de funções para a manipulação das entradas e saídas. Por isso, não podemos esquecer de utilizá-la em nossos programas. Logo abaixo, temos a função **main(){ ... }**, que é a primeira a ser executada quando o programa é executado, é nesse local que todo o nosso bloco de códigos é inserido. Dentro da função **main()**, temos a função **printf()**, esta faz que os elementos sejam impressos na tela tela do computador.

Após a compilação do código fonte, podemos executá-lo e o resultado será o apresentado na Figura 1.3.



1FIGURA 3.59 - Programa "meu primeiro código" em execução FONTE: Dev-C ++

Podemos compreender que a estrutura básica de um código fonte desenvolvido em linguagem C deve conter elementos básicos como:

1FIGURA 4.59 - Estrutura código em linguagem C FONTE: Dev-C ++.

Variáveis

Para que possamos efetuar a manipulação de dados, devemos utilizar variáveis. Mas o que é uma variável, afinal? Segundo Mizrahi (2008, p. 13):

> As variáveis são o aspecto fundamental de qualquer linguagem de computador. Uma variável em C é um espaço de memória reservado para armazenar um certo tipo de dado e tendo um nome para referenciar o seu conteúdo. [...] uma variável é um espaço de memória que pode contar, a cada tempo, valores diferentes.

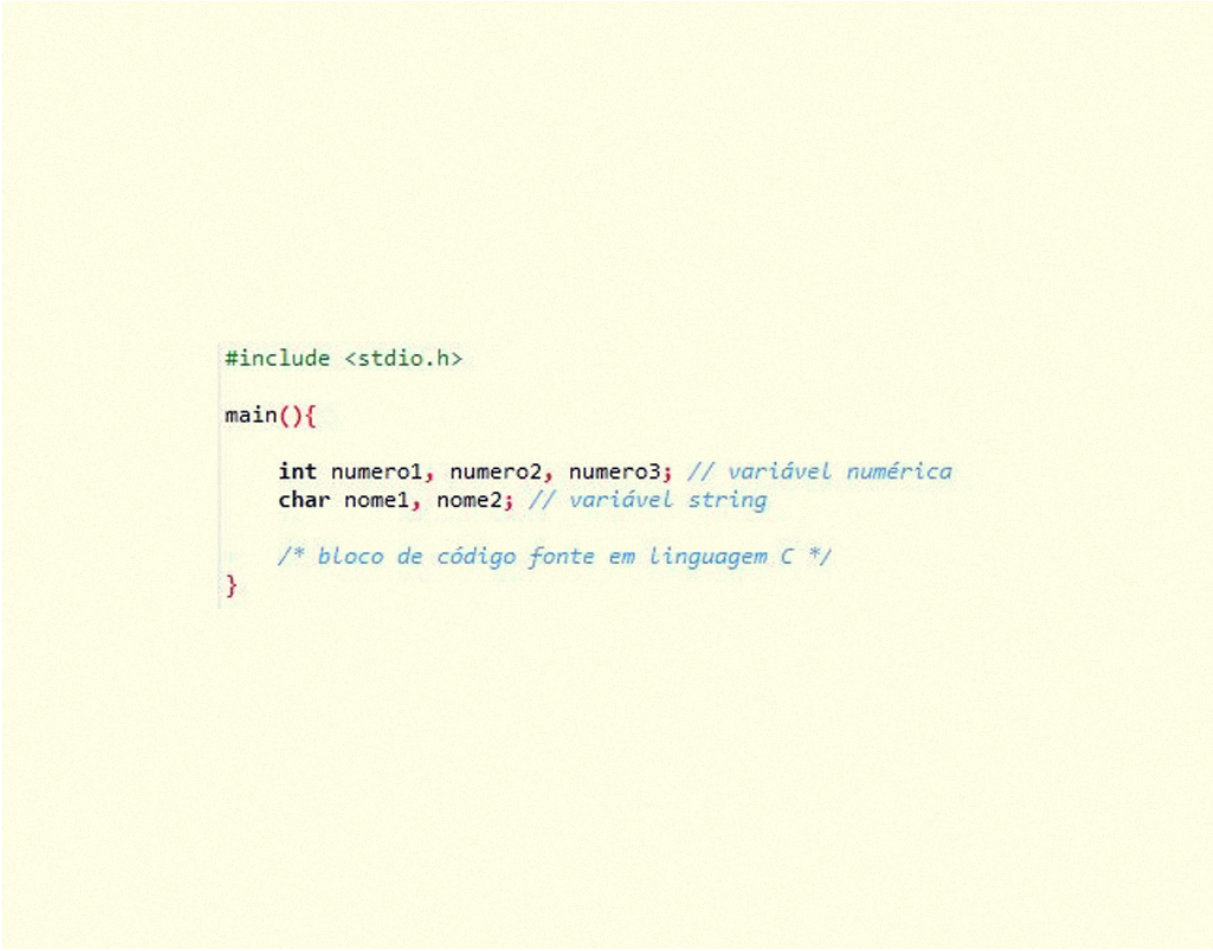
Para utilizar as variáveis, devemos declará-las no bloco de códigos, sendo necessário informar o seu tipo e um nome de referência, o espaço de memória a ser reservado é determinado com base no tipo da variável, podendo ser dados números ou caracteres (strings).

Vejamos na Figura 1.5 como efetuar a declaração de variáveis:

```c
#include <stdio.h>

main(){

    int numero1; // variável numérica
    int numero2; // variável numérica
    char nome1; // variável string
    char nome2; // variável string

    /* bloco de código fonte em linguagem C */
}
```

1FIGURA 5.59 - Estrutura de declaração de variáveis FONTE: Dev-C ++.

Quando temos diversas variáveis com o mesmo tipo, podemos declarar todas na mesma linha separando apenas por uma vírgula, conforme a Figura 1.6:



```c
#include <stdio.h>

main(){

    int numero1, numero2, numero3; // variável numérica
    char nome1, nome2; // variável string

    /* bloco de código fonte em linguagem C */
}
```

1FIGURA 6.59 - Declaração de variáveis FONTE: Dev-C ++.

Um ponto muito importante na declaração de uma variável é que os nomes de referência não podem começar com números, não podem conter caracteres especiais ou espaços. Veja no Quadro 1.1 algumas variáveis declaradas de modo incorreto:

DECLARAÇÃO DE VARIÁVEL	
int 1numero; **int** 55ponto, 53ponto; **char** 1nome; **char** 1lugar, 2lugar;	Variáveis declaradas de modo incorreto, pois o nome de referência se inicia com número.
int número; **int** número, código; **char** endereço; **char** endereço, município;	Variáveis declaradas de modo incorreto,pois o nome de referência contém caracteres especiais.
int número de telefone; **int** número de telefone, código do produto; **char** primeiro nome; **char** primeiro nome, endereço residencial;	Variáveis declaradas de modo incorreto, pois o nome de referência contém caracteres especiais e espaços.

1QUADRO 1.9 - undefined

Declaração de variáveis de modo incorreto

o autor.

Vejamos no Quadro 1.2 como é possível declarar as variáveis anteriormente informadas de modo correto:

DECLARAÇÃO DE VARIÁVEL	
int numero1; **int** ponto55, ponto55; **char** nome1; **char** lugar1, lugar2;	Variáveis declaradas de modo correto, pois o nome de referência não se inicia com número.
int numero; **int** numero, codigo; **char** endereco; **char** endereco, municipio;	Variáveis declaradas de modo correto, pois o nome de referência não contém caracteres especiais.
int numero_de_telefone; ou **int** numerodetelefone; **int** numero_de_telefone, codigo_do_produto; **char** primeiro_nome; **char** primeiro_nome, endereco_residencial;	Variáveis declaradas de modo correto, pois o nome de referência não contém caracteres especiais ou espaços (podemos substituir os espaços por underline).

1QUADRO 2.9 - undefined

Declaração de variáveis de modo correto

o autor.

Tipos de Variáveis

Na linguagem C, os tipos de variáveis mais utilizados são: **char**, **int**, **float**, **double**, **enum** e **pointer.**

O tipo **char** é um string (composto por caracteres), podendo conter palavras e números. O tipo **int** é numérico inteiro, armazena apenas números não fracionários. O tipo **float** é numérico de ponto flutuante, aceitando números decimais e fracionários. O tipo **double** é de ponto flutuante de precisão dupla, muito parecido com o **float**, mas apresenta maior capacidade de armazenamento. O tipo **enum** representa um lista predefinida pelo desenvolvedor. O tipo **pointer** é um tipo especial, pois não armazena dados, mas sim a localização na memória de um dado real.

TIPO	DADOS ACEITOS	TAMANHO EM BYTES
char	letras, caracteres especiais e números	1
int	números inteiros de -32767 até 32767	4
float	números decimais de -3.4×10^{38} até $+3.4 \times 10^{+38}$, até 6 dígitos	4
double	números decimais de -1.7×10^{308} até $+1.7 \times 10^{+308}$ até 10 dígitos	8

1QUADRO 3.9 - undefined

Tipos de variáveis

o autor.

¶ Reflita

Ao analisarmos os tipos de variáveis, é possível compreender que cada dado terá uma declaração específica. Quando não sabemos quais serão as informações a serem manipuladas, podemos dizer que será mais seguro se todos os tipos de variáveis sejam char?

Nesse caso, tenha como base a informação de que esse tipo de variável aceita letras, caracteres especiais e números.

Conforme informado anteriormente, a declaração de variáveis acontece entre o **#include** e a função **main()**. Veja na Figura 1.7 como é possível efetuar esse procedimento.

```c
#include <stdio.h>

////////// declaração de variáveis //////////

char nome_completo; /* String */
int idade; /* numéro inteiro */
float peso; /* numéro fracionário normal */
double altura; /* numéro fracionário grande */
int *ptr;  // declara um ponteiro para um inteiro

/*Aqui os valores azul = 4 e roxo = 5 são incrementados automaticamente*/
enum cores { amarelo = 3, azul, roxo };

////////// declaração de variáveis //////////

main(){
    //bloco de código;
}
```

1FIGURA 7.59 - Declaração de variáveis no corpo do código fonte FONTE: Dev-C ++.

Palavras Reservadas

Todos os tipos de linguagens têm palavras reservadas, elas não podem ser utilizadas para declaração de variáveis, funções ou na manipulação de dados. Cada palavra reservada representa um conjunto de condições e tem seu uso específico. Na linguagem C, podemos citar como exemplo as palavras: **operator**, **case**, **void**, **return**. Ao todo, na linguagem C, temos 32 palavras chaves, no Quadro 1.4, são apresentadas algumas delas:

PALAVRAS RESERVADAS			
asm	tamplate	do	register
catch	this	double	return
class	virtual	else	short
delete	_cs	enum	signed
_export	_ds	extern	sizeof
frient	_es	far	static
inline	_ss	float	struct
_loadds	auto	for	switch
new	break	goto	typedef
operador	case	huge	union
private	catch	if	unsigned
protected	cdecl	int	void
public	char	interrupt	volatile
_regparam	const	long	while
_saveregs	continue	near	
_seg	default	pascal	

1QUADRO 4.9 - undefined

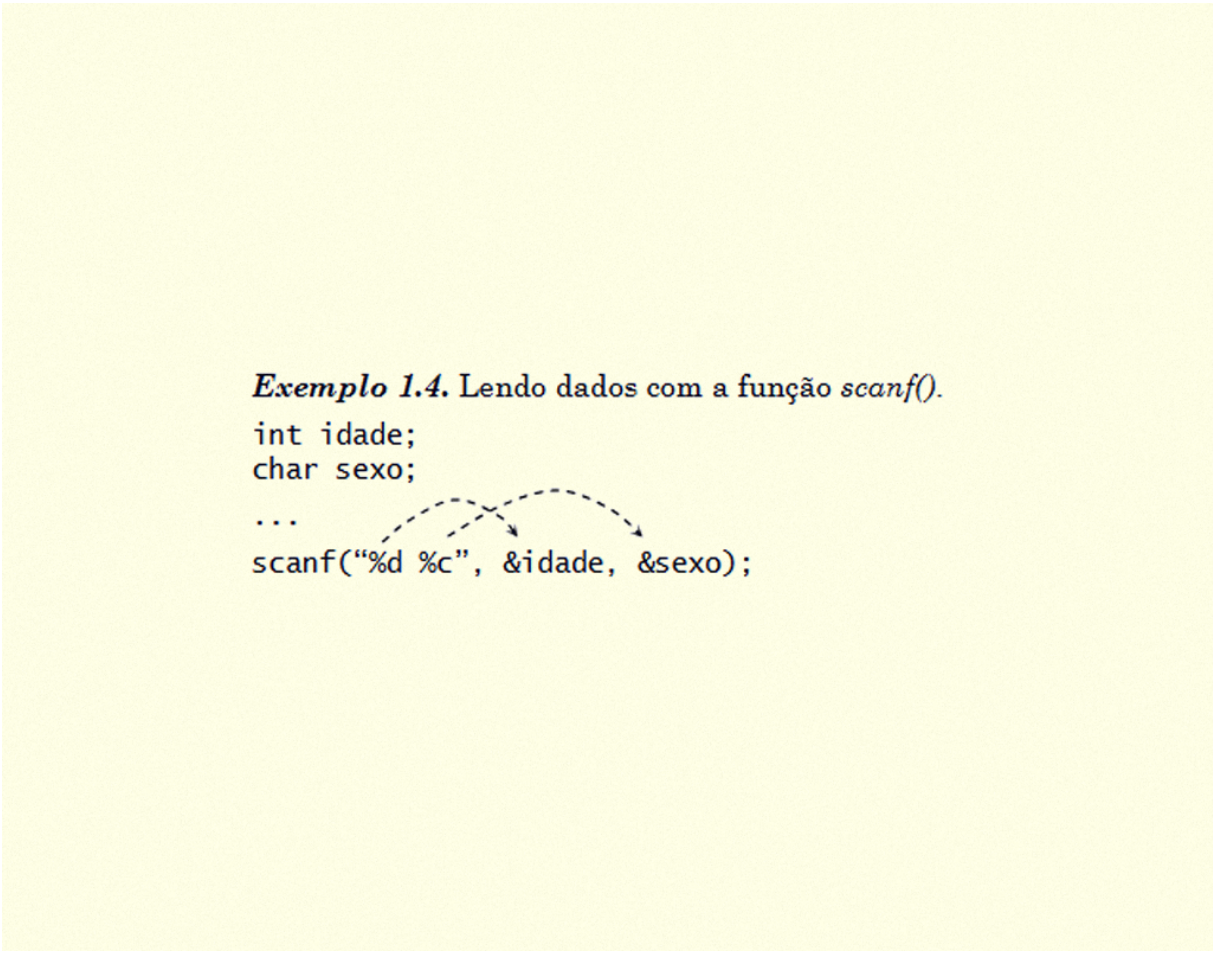
Palavras reservadas da linguagem C

Pappas e Murray (1991).

Entrada e Saída de Dados

Um dos objetivos principais de um programa, sendo ele em C ou qualquer outra linguagem, é a manipulação de informações, esse processo se inicia com a entrada de dados, os quais podem ser variáveis predefinidas no programa ou conteúdo digitado pelo usuário. A linguagem C utiliza a função **scanf()** para obter os dados digitados pelo usuário no programa.

> A função scanf() permite que um valor seja lido do teclado e armazenado numa variável. Sua sintaxe consiste numa cadeia de formatação seguida de uma lista de argumentos, cada um deles sendo o endereço de uma variável .
>
> (PEREIRA, 2001, p.6)



1FIGURA 8.59 - Lendo dados com a função scanf() FONTE: Pereira (2001, p. 6).

Observe que, para efetuar a leitura da entrada de dados, à frente do nome da variável, temos o caractere & (e comercial), em que se faz a indicação do espaço reservado na memória, sem o caractere o valor não é atribuído para a variável. Esse caractere é utilizado apenas na entrada de dados, para utilizar a variável em outras partes do código fonte, não é necessário utilizá-lo.

Ao efetuar a leitura de dados, temos uma formatação específica para cada tipo de variável, no exemplo apresentado, estão sendo utilizadas duas variáveis de tipos diferentes: **%d** manipula dados inteiros, e **%c** está manipulando dados de um string. No Quadro 1.5, podemos verificar outros tipos de formatação.

EXPRESSÃO	FORMATAÇÃO ACEITA
%d	Número decimal inteiro.
%c	Único caractere.
%u	Decimal sem sinal.
%i	Decimal inteiro.
%e	Número em ponto flutuante com sinal opcional.
%f	Número em ponto flutuante com ponto opcional.
%g	Número em ponto flutuante com expoente opcional.
%p	Ponteiro.
%s	String.
%o	Número em base octal.

%x	Número em base hexadecimal.

1QUADRO 5.9 - undefined

Expressão para manipulação de tipos de variáveis

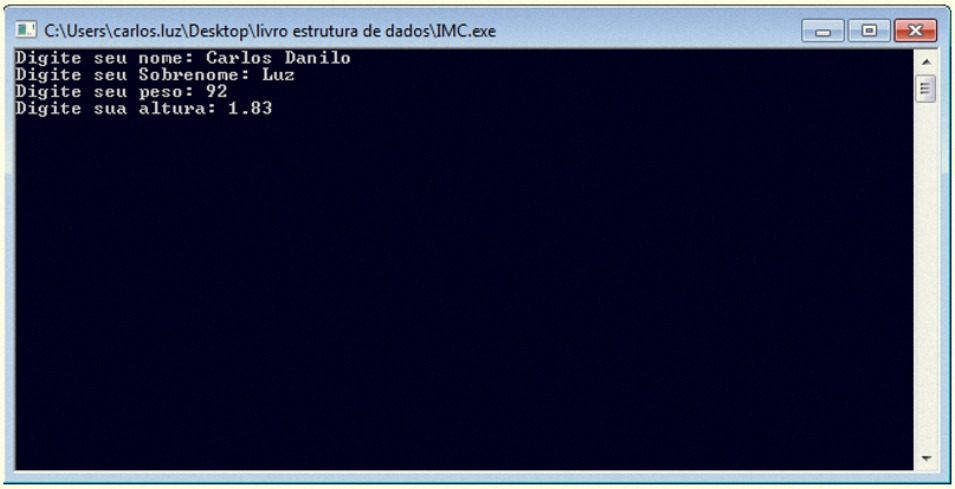
o autor.

Para compreendermos melhor a utilização dessa função, vejamos o código fonte apresentado na Figura 1.9.

```c
#include <stdio.h>

char nome[30], sobnome[30]; // armazena até 30 caracteres
float peso, altura;

main(){

    printf("Digite seu nome: ");
    scanf("%[^\n]s", &nome); // armazena a entrada de dados com espaço

    printf("Digite seu Sobrenome: ");
    scanf("%s", &sobnome);

    printf("Digite seu peso: ");
    scanf("%f", &peso);

    printf("Digite sua altura: ");
    scanf("%f", &altura);

}
```

1FIGURA 9.59 - Código fonte, leitura de variáveis FONTE: Dev-C ++.

Após compilar o código fonte e executar o programa, o usuário irá digitar o que é requisitado, conforme Figura 1.10.



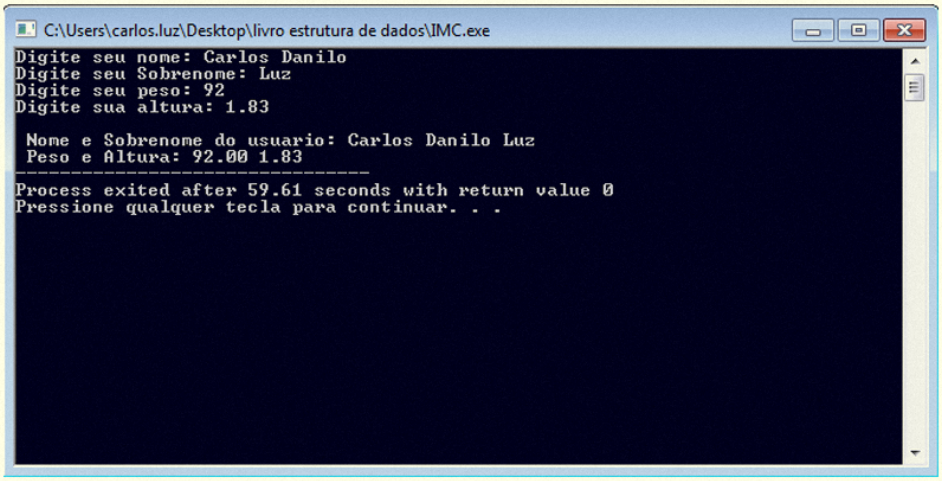
1FIGURA 10.59 - Resultado do código fonte, sobre leitura de variáveis FONTE: Dev-C ++.

A saída de dados ou impressão na tela, como já foi citado anteriormente, se utiliza a função **printf( )**. Pereira (2001) afirma que "***a função printf( ) nos permite exibir informações formatadas no vídeo. A sua sintaxe é essencialmente idêntica àquela da função scanf().***" Podemos compreender que as expressões de entrada de dados (**%d**, **%f**, **%s**) são utilizadas em ambas as funções, conforme apresentado na Figura 1.11.

```c
#include <stdio.h>

char nome[30], sobnome[30]; // armazena até 30 caracteres
float peso, altura;

main(){

    printf("Digite seu nome: ");
    scanf("%[^\n]s", &nome); // armazena a entrada de dados com espaço

    printf("Digite seu Sobrenome: ");
    scanf("%s", &sobnome);

    printf("Digite seu peso: ");
    scanf("%f", &peso);

    printf("Digite sua altura: ");
    scanf("%f", &altura);

    printf("\n Nome e Sobrenome do usuario: %s %s", nome, sobnome);
    printf("\n Peso e Altura: %0.2f %0.2f", peso, altura);

}
```

1FIGURA 11.59 - Código fonte com itens de leitura e escrita em linguagem C FONTE: Dev-C ++.

Resultado do programa em execução:



1FIGURA 12.59 - Resultado do código fonte, sobre leitura e escrita FONTE: Dev-C ++

Expressões e Operadores

A linguagem C se utiliza de um conjunto de expressões e operadores para efetuar funções matemáticas ou relacionais, são usados vários caracteres especiais para representação.

No Quadro 1.6, são apresentados os operadores aritméticos em C.

OPERADOR	EXEMPLO	COMENTÁRIO
=	x = y	O conteúdo da variável Y é atribuído à variável X (a uma variável pode ser atribuído o conteúdo de outra, um valor constante ou, ainda, o resultado de uma função).
+	x + y	Soma o conteúdo de X e de Y.
-	x - y	Subtrai o conteúdo de Y do conteúdo de X.
*	x * y	Multiplica o conteúdo de X pelo conteúdo de Y.

/	X / y	Obtém o quociente de divisão de X por Y. Se os operadores são inteiros, o resultado da operação será o quociente inteiro da divisão. Se os operadores são reais, o resultado da operação será a divisão. Por exemplo: int z = 5/2: a variável z receberá o valor 2 float z = 5/2: a variável z receberá o valor de 2,5
%	x % y	Obtém o resto da divisão de X por Y.

1QUADRO 6.9 - undefined

Operadores aritméticos em C

Ascencio e Campos (2012 p. 29).

Expressões **relacionais** referem-se a comparações ou validações entre dois dados e se utilizam de símbolos próprios. Através do Quadro 1.7, podemos compreender essas expressões:

OPERADOR	EXEMPLO	COMENTÁRIO
==	X == Y	O conteúdo de X é igual ao conteúdo de Y.
!=	X != Y	O conteúdo de X é diferente do conteúdo de Y.
<=	X <= Y	O conteúdo de X é menor ou igual ao conteúdo de Y.
>=	X >= Y	O conteúdo de X é maior ou igual ao conteúdo de Y.
<	X < Y	O conteúdo de X é menor que o conteúdo de Y.
>	X > Y	O conteúdo de X é maior que o conteúdo de Y.

1QUADRO 7.9 - undefined

Expressões de comparação

Ascencio e Campos (2012 p.30).

Os operadores **matemáticos de atribuição** são utilizados para representar de maneira sintética uma operação aritmética e, posteriormente, uma operação de atribuição (ASCENCIO; CAMPOS, 2012 p.29). Esses operadores são uma combinação das expressões aritméticas e relacionais, veja-os descritos no Quadro 1.8:

OPERADOR	EXEMPLO	COMENTÁRIO
+=	X += Y	Equivalente a X = X + Y
-=	X -= Y	Equivalente a X = X - Y
*=	X *= Y	Equivalente a X = X * Y
/=	X /= Y	Equivalente a X = X / Y
%=	X %= Y	Equivalente a X = X % Y
++	X++	Equivalente a X = X +1
++	Y = ++ X	Equivalente a X = X + 1 e depois Y = X
++	Y = X++	Equivalente a Y = X e depois X = X + 1
--	X--	Equivalente a X = X -1
--	Y = -- X	Equivalente a X = X -1 e depois Y = X
--	Y = X --	Equivalente a Y = X e depois X = X -1

1QUADRO 8.9 - undefined

Operadores Matemáticos de combinações

Ascencio e Campos (2012, p. 29).

¶ Fique por dentro

Em linguagem C, para atribuir um dado/valor a uma variável, se utiliza apenas um igual =, ao efetuarmos uma comparação entre dois valores/dados, utilizamos dois iguais ==, esse conceito se aplica para as demais linguagem de programação conhecidas.

A linguagem C oferece operadores lógicos, que podem ser usados para formar condições mais complexas ao combinar condições simples. Os operadores lógicos são && (AND lógico), II (OR lógico) e !(NOT lógico, também chamado de negação lógica) (DEITEL; DEITEL. 2011 p.95).

O retorno de uma expressão lógica é verdadeiro ou falso, tendo como base a tabela verdade utilizada em matemática para computação. No quadro 1.9 são descritos os operadores lógicos:

OPERADORES	TIPO
< <= > >=	Relacional
== !=	Igualdade
&&	AND lógico
||	OR lógico
?:	condicional
= += -= *= /= %=	Atribuição

1QUADRO 9.9 - undefined

Operadores Lógicos

adaptado de Deitel e Deitel (2011, p.97).

Instruções Condicionais e de Repetição

As expressões condicionais fazem o tratamento dos dados inseridos pelo usuário, efetuando comparações ou validações, de forma que o programa execute parte do código ou não.

> Uma das tarefas fundamentais de qualquer programa é decidir o que deve ser executado e seguir. Os comandos de decisão permitem determinar qual é a ação a ser tomada com base no resultado de um expressão condicional. Isso significa que podemos selecionar entre ações alternativas, dependendo de critérios desenvolvidos no decorrer da execução do programa
>
> (MIZRAHI, 2008 p.84)

Vamos estudar três tipos expressões:

- Simples (if).
- Composta (if-else).
- Múltipla Escolha (switch-case) .

Em conjunto com as expressões condicionais, temos as estruturas de repetição, também chamadas de laços, essas são utilizadas quando temos que efetuar várias entradas de dados ou executar condições.

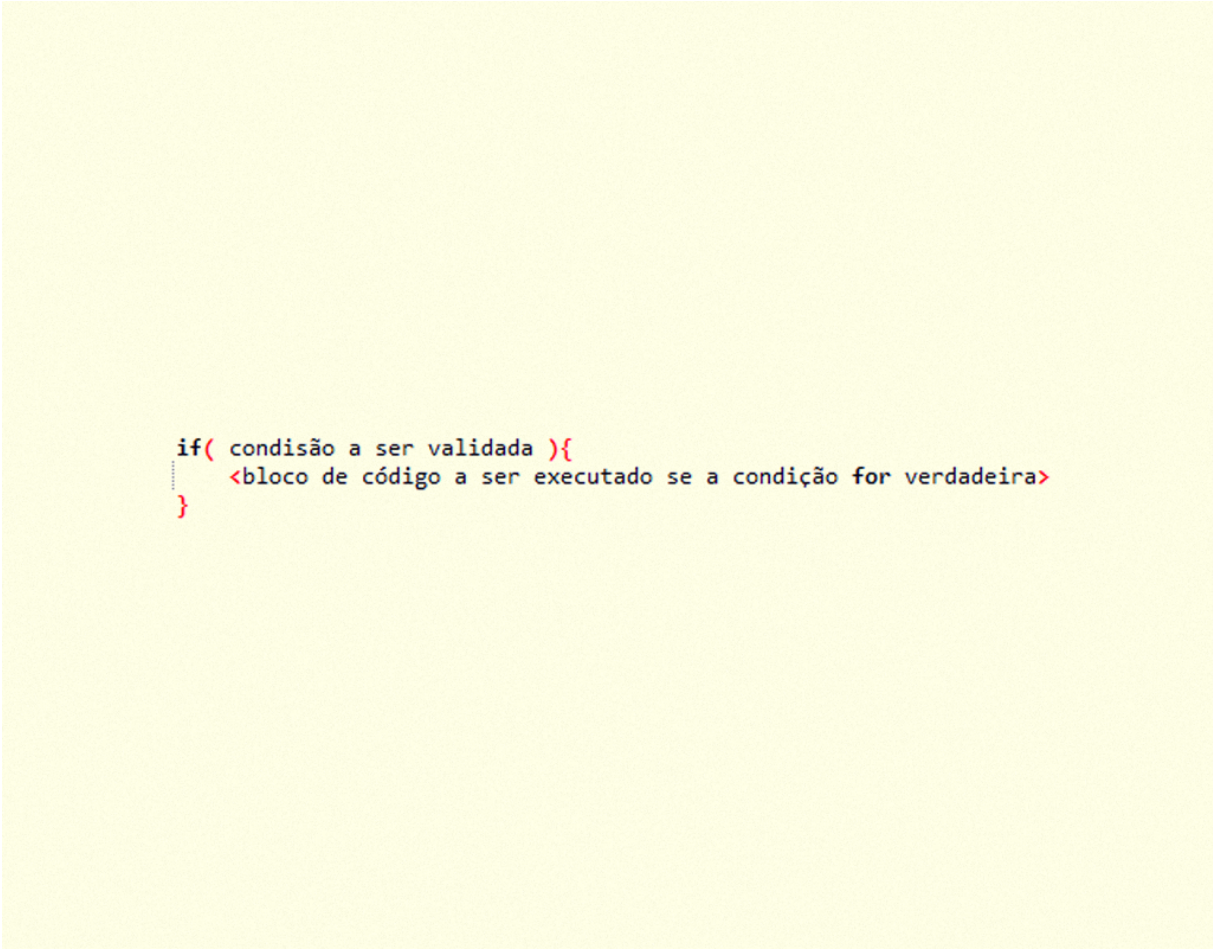
Uma estrutura de repetição permite que você especifique que uma ação deverá ser repetida enquanto alguma condição permanecer verdadeira (DEITEL, 2011 p.52). Pode-se compreender como expressão verdadeira os exemplo de laços:

- FOR: faça a repetição enquanto não tiver o nome de 10 alunos.
- WHILE: irá fazer uma comparação para iniciar o laço, se não tiver os nomes dos aluno, caso iniciado o laço, só termina se tivermos os 10 nomes.
- DO - WHILE: executa o laço pelo menos uma vez, após iniciado só para se obter os 10 nomes de alunos.

No decorrer desta unidade, vamos compreender melhor cada um dos tipos de laços de repetição.

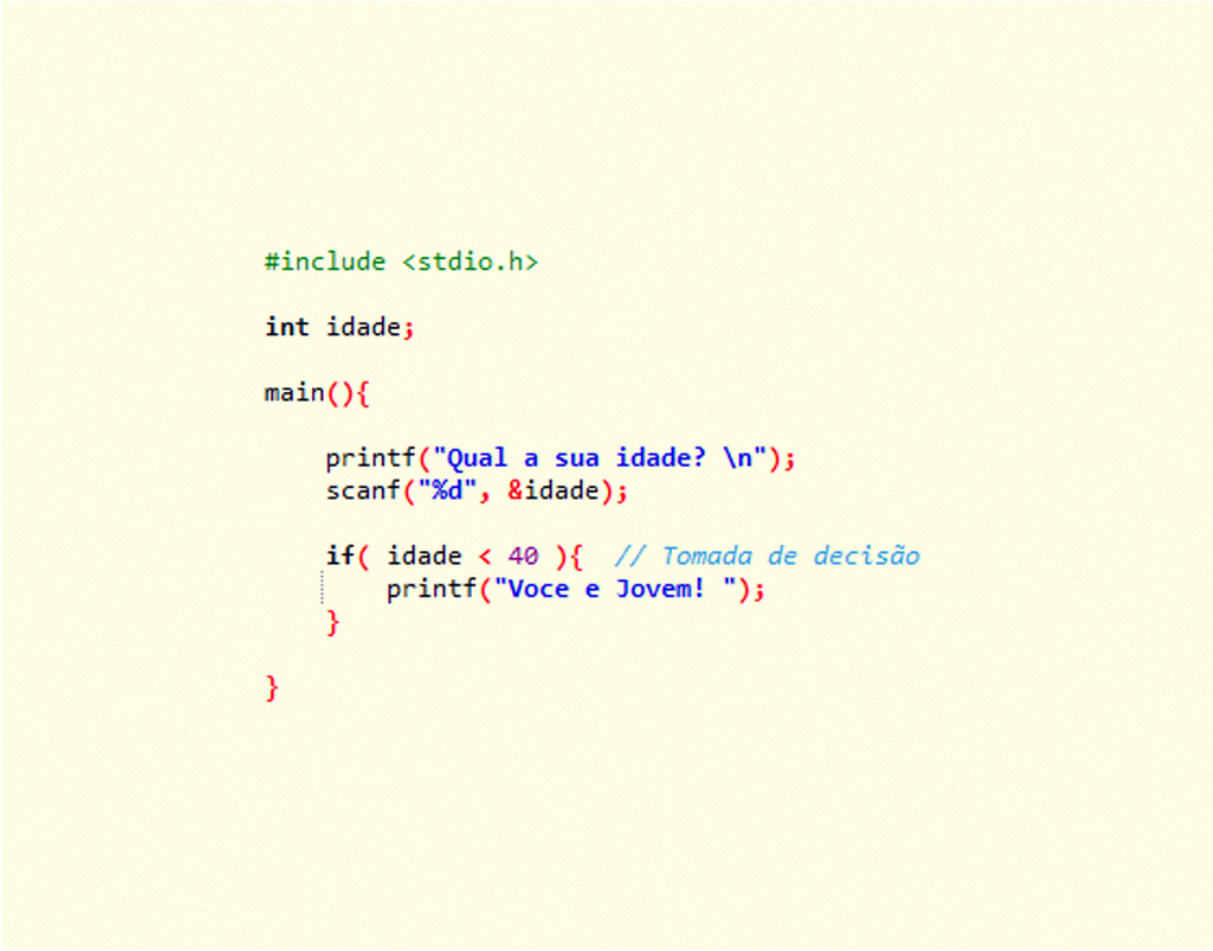
Instrução Condicional Simples

A expressão condicional simples efetua a comparação entre os dados informados pelo usuário para assim tomar uma decisão. De modo que, se a condição retornar verdadeira, o bloco de código é executado.



1FIGURA 13.59 - Comando condicional simples FONTE: o autor.

Para que o código seja executado após a tomada de decisão, ele deve estar entre { ... }. A primeira chave, este símbolo { , marca o início do código e, após ela, todos comandos serão executados, o término da condição é identificado por uma segunda chave que fecha o código, com este símbolo: }. Toda estrutura condicional tem seu começo, meio e fim. Caso a condição seja falsa, o bloco de código não será executado.



```c
#include <stdio.h>

int idade;

main(){

    printf("Qual a sua idade? \n");
    scanf("%d", &idade);

    if( idade < 40 ){  // Tomada de decisão
        printf("Voce e Jovem! ");
    }

}
```

1FIGURA 14.59 - Limitações de execução FONTE: DEV C ++.

Ao analisarmos o código fonte, podemos compreender que, se o usuário informar que sua idade é de até 39 anos, o programa imprime na tela a frase "Voce e Jovem !", pois a condição é verdadeira, caso o usuário tivesse digitado 41, não seria impresso nada na tela. Podemos utilizar expressões e operadores para efetuar a comparação de duas ou mais condições, tendo como base a tabela verdade. A estrutura do comando continua a mesma, apenas inserindo operadores da tomada de decisão, podemos também ter no mesmo código várias condições se utilizando da mesma variável para a tomada de decisão.

```c
#include <stdio.h>

int idade;

main(){

    printf("Qual a sua idade? \n");
    scanf("%d", &idade);

    if( idade >= 20 && idade <= 30 ){  // Tomada de decisão
        printf("Voce e Jovem! ");
    }

    if( idade > 30 && idade <= 40 ){  // Tomada de decisão
        printf("Voce e nao e tao Jovem! ");
    }

}
```

1FIGURA 15.59 - Exemplo de comando com duas condicionais FONTE: DEV C ++.

Ao executar o programa, é possível tomar duas decisões, com base na idade informada, se utilizando de instruções condicionais simples, expressões e operadores.

¶ Fique por dentro

Para efetuar as comparações de mais elementos na mesma condicional, utilizamos como base o conceito de tabela verdade.

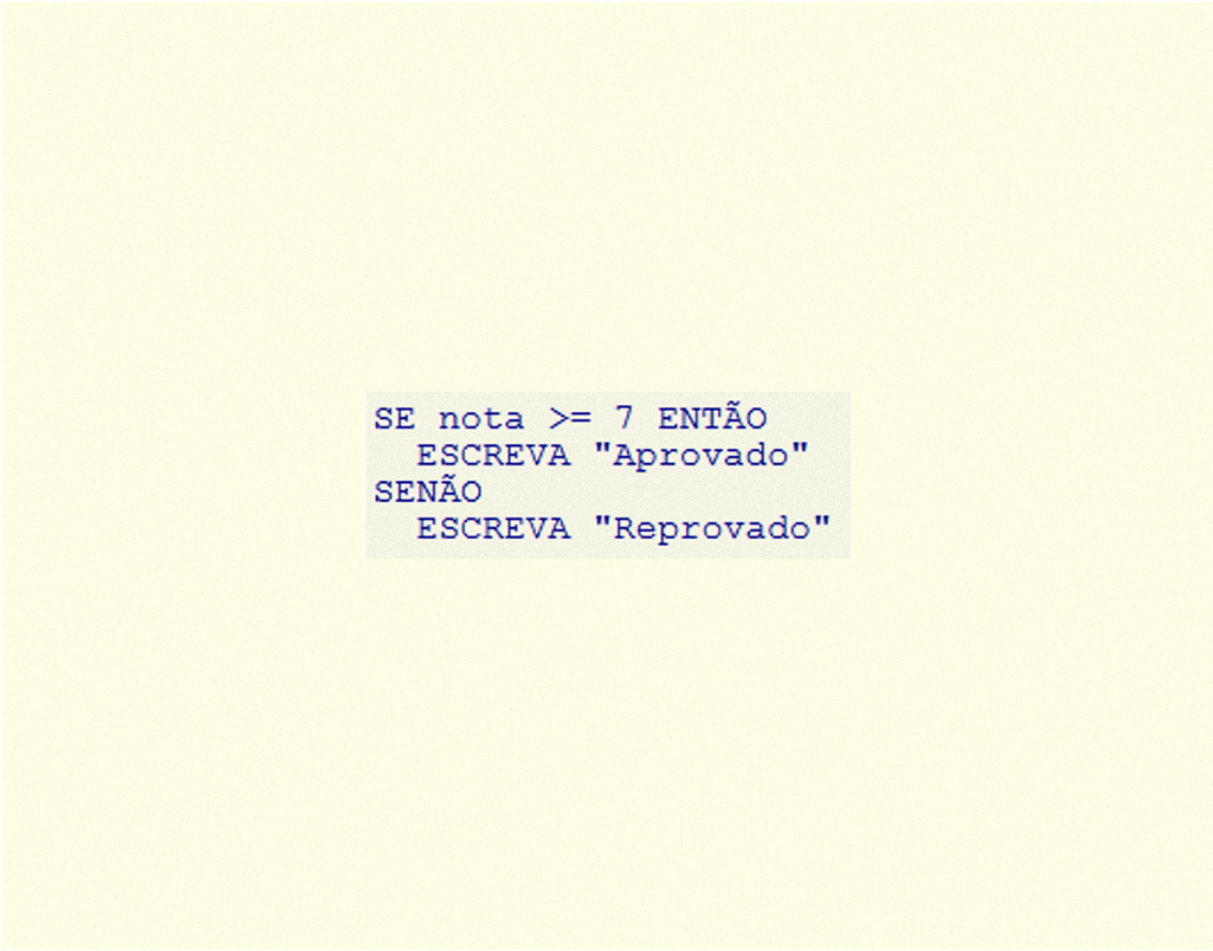
FONTE o autor.

TABELA VERDADE			
Objeto A	Objeto B	A e B	A ou B

Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso	Verdadeiro
Falso	Verdadeiro	Falso	Verdadeiro
Falso	Falso	Falso	Falso

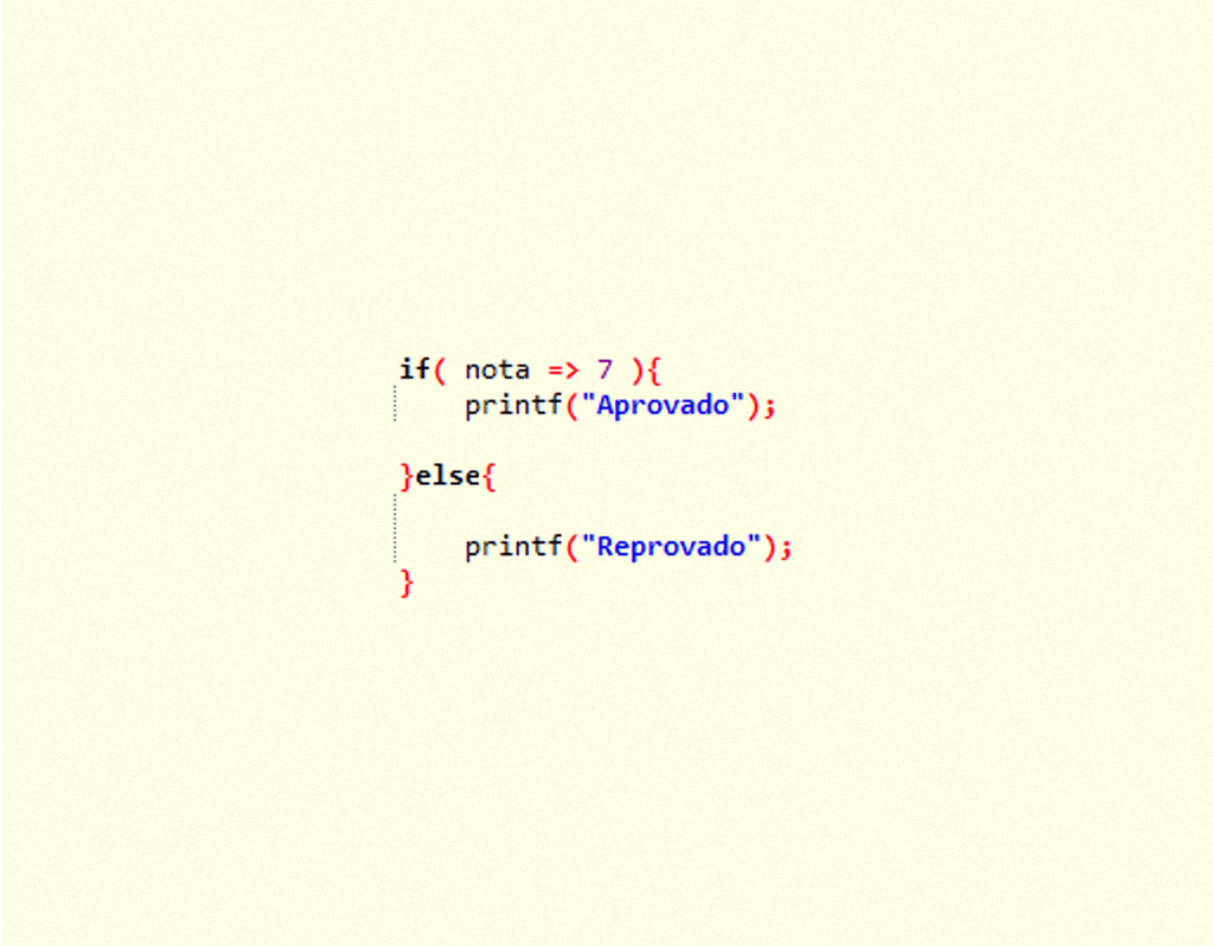
Instrução Condicional Composta

A instrução condicional composta seria uma extensão da simples, pois temos a mesma base de estrutura de código, agora, caso a primeira condição não retornar verdadeira, o conteúdo é direcionado para a outra situação. Ao recordar do pseudocódigo, temos as expressões SE e SENÃO:



1FIGURA 16.59 - Representação textual em pseudocódigo, sobre condicional composta FONTE: o autor.

O pseudocódigo apresentado, ao ser executado, efetua uma tomada de decisão sobre a nota de uma aluno, a qual se for maior ou igual que 7, torna verdadeira a condição, assim, será impresso na tela "Aprovado", caso contrário, será impresso "Reprovado". A estrutura do código em linguagem C segue o mesmo princípio desta expressão, dando continuidade à tomada de decisão simples:



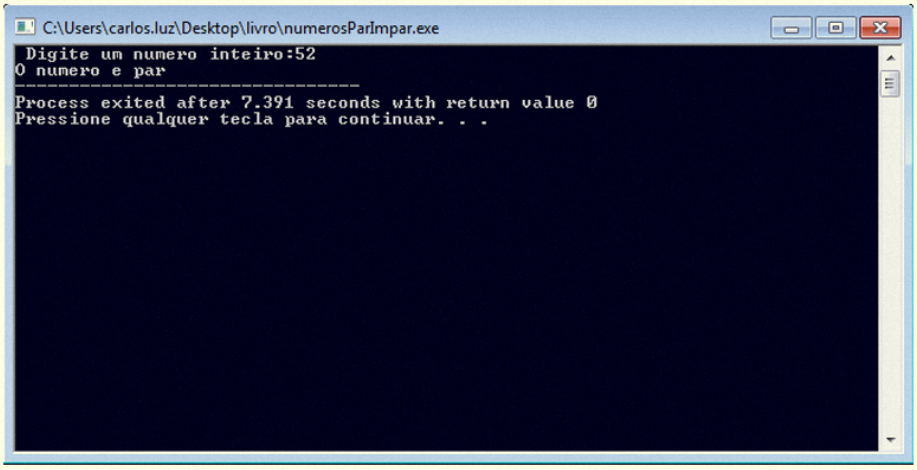
1FIGURA 17.59 - Código de exemplo condicional composta FONTE: DEV C ++.

No Figura 1.18, temos o código fonte de um programa em C, em que o usuário informa um número e é retornado na tela se este é par ou ímpar, podemos utilizar a expressão % 2, na comparação do if, que retorna o resto da divisão, se o retorno for 0, ele é par, se não, é ímpar.

```c
#include <stdio.h>

int num;

main (){

    printf(" Digite um numero inteiro:");
    scanf ("%d", &num);

    if (num % 2 == 0 ){
        printf ("O numero e par");

    }else{
        printf ("O numero e impar");

    }
}
```

1FIGURA 18.59 - Programa de exemplo com condicional composta FONTE: DEV C ++.

Ao executarmos o programa e inserirmos algum número, o programa efetua o procedimento e, assim, temos a seguinte resposta:



1FIGURA 19.59 - Resultado do programa em execução, condicional composta FONTE: DEV C ++.

Instrução Condicional Múltipla Escolha

As estruturas condicionais simples e compostas são úteis em casos em que a tomada de decisão tem que escolher 2 caminhos, com base na expressão ou operador.

> C tem um comando interno de seleção múltipla, **switch**, que testa sucessivamente o valor de uma expressão contra uma lista de constantes internas ou de caractere. Quando o valor coincide, os comandos associados àquela constante são executados.
>
> (SCHILDT, 1997 p. 70)

Podemos compreender que a linguagem C irá percorrer um lista de **Case** até encontrar a que seja a resposta de busca. Após encontrar a constante, o programa executa a sequência de código e, em seguida, a função **break**, que seria o comando de desvio sinalizando que o objeto já foi encontrado, não havendo mais a necessidade de continuar o switch case.

Pode-se utilizar o comando **default**, que é identificado como sendo a saída padrão, caso nenhum dos **case** atenda a necessidade, o bloco de código a ser executado será o pertencente ao **default**. Vamos ver como são escritos os comando em linguagem C:

```
switch (expressão){

    case condição1:
        <bloco de códigos>
        break;

    case condição2:
        <bloco de códigos>
        break;

    case condição3:
        <bloco de códigos>
        break
    .
    .
    .
    default:
        <bloco de código padrão>
}
```

1FIGURA 20.59 - Composição do SWITCH CASE de forma textual FONTE: o autor.

Schildt (1997 p. 70-71) aponta três itens importantes sobre o **switch-case**:

O comando **switch** difere do comando **if** porque **switch** só pode testar igualdade, enquanto **if** pode avaliar uma expressão lógica ou relacional.

Duas constantes **case** no mesmo **switch** não podem ter valores idênticos. Obviamente, um comando **switch** incluído em outro switch mais externo pode ter as mesmas constantes **case**.

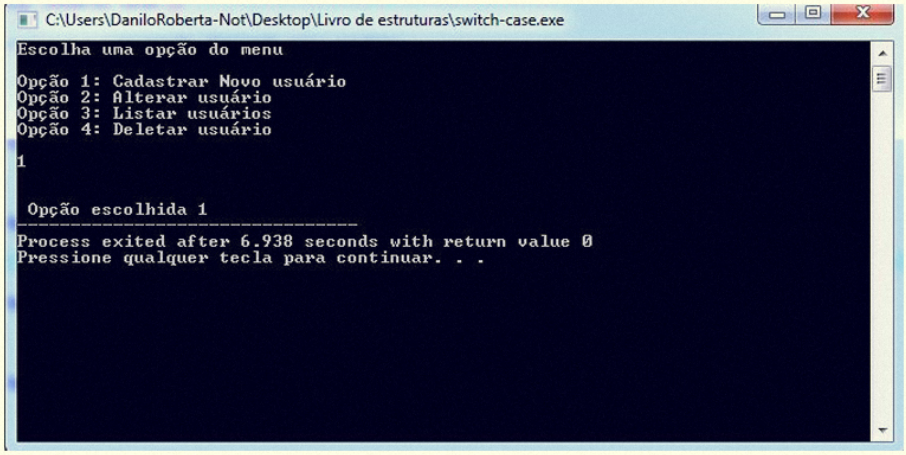
Se constantes de caractere são usadas em um comando switch, elas são automaticamente convertidas para seus valores inteiros.

Esses três pontos são as regras para se utilizar a instrução condicional case, para compreendermos melhor vamos analisar o código fonte a seguir:

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

int opcao;

main(){
    setlocale(LC_ALL, "Portuguese");//habilita a acentuação para o português

    printf("Escolha uma opção do menu \n\n");
    printf("Opção 1: Cadastrar Novo usuário \n");
    printf("Opção 2: Alterar usuário \n");
    printf("Opção 3: Listar usuários \n");
    printf("Opção 4: Deletar usuário \n\n");

    opcao=getchar(); // capitura a informação digitada pelo usuário

    switch (opcao){

        case  1 :
            printf("\n\n Opção escolhida 1");
            break;

        case  2 :
            printf("\n\n Opção escolhida 2");
            break;

        case  3 :
            printf("\n\n Opção escolhida 3");
            break;

        case  4 :
            printf("\n\n Opção escolhida 4");
            break;

        default:
            printf("\n\n Nenhuma opção válida");
    }

}
```

1FIGURA 21.59 - Código fonte de exemplo, utilizando SWITCH CASE FONTE: DEV C ++

Ao compilarmos o código fonte apresentado, o usuário deverá digitar o número de uma opção (1, 2, 3, 4), em seguida, será executado o switch-case, efetuando as comparações. Caso a informação digitada seja compatível com alguma das opções, será impressa na tela a informação pertinente, caso não seja encontrada igualdade, a opção default será impressa na tela.

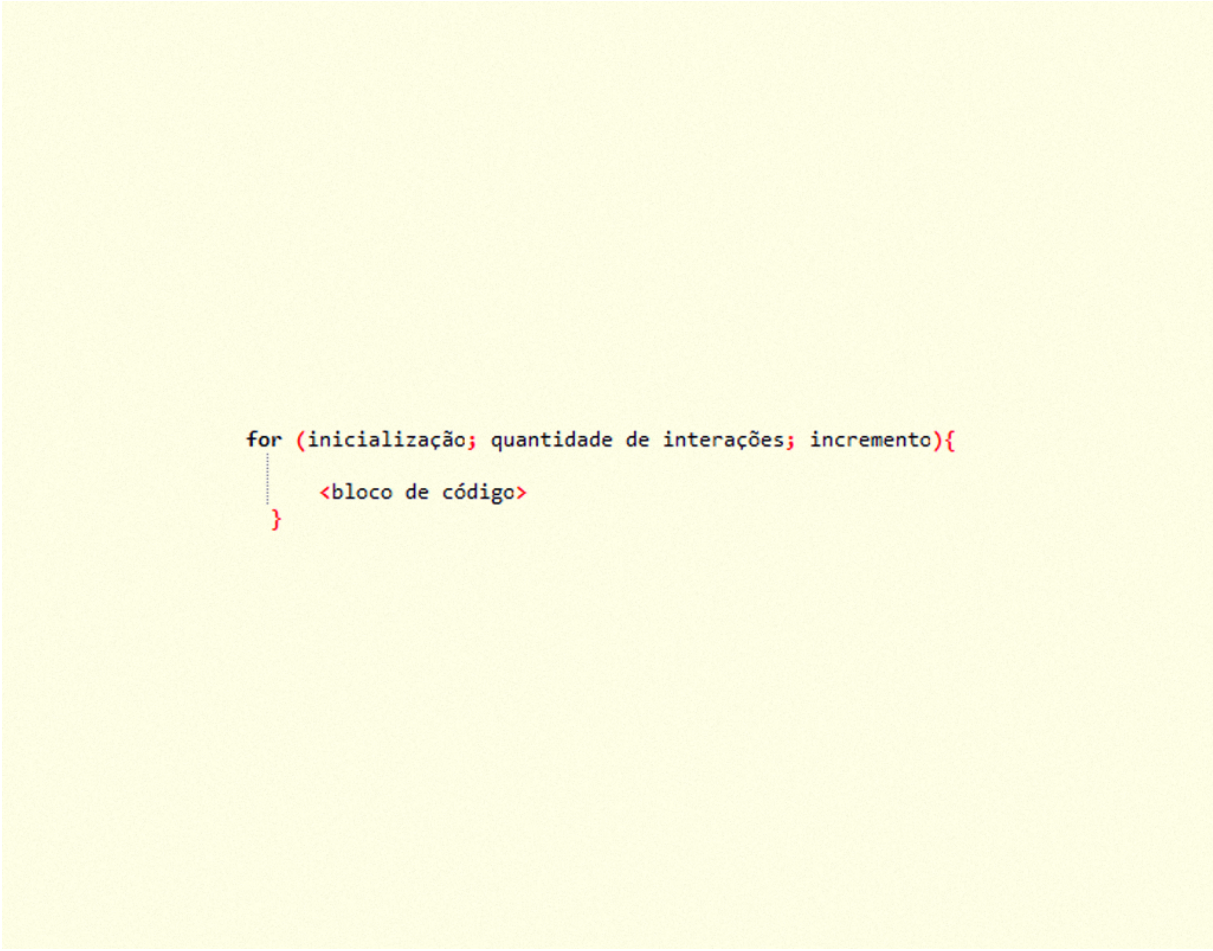


1FIGURA 22.59 - Resultado do programa em execução FONTE: DEV C ++.

Estrutura de Repetição FOR

Em linguagem C e em todas as outras linguagens modernas de programação, os comandos de iteração (também chamados laços) permitem que um conjunto de instruções seja executado até que ocorra certa condição (SCHILDT, 1997, p.74).

Os laços de repetição efetuam as iterações até que uma condição seja atendida, no caso do comando **for**, é informado a variável de inicialização, a condição de iteração e o comando de incremento, cada item separado por ponto e vírgula - ;. Veja:



1FIGURA 23.59 - Composição do comando FOR de modo textual FONTE: o autor

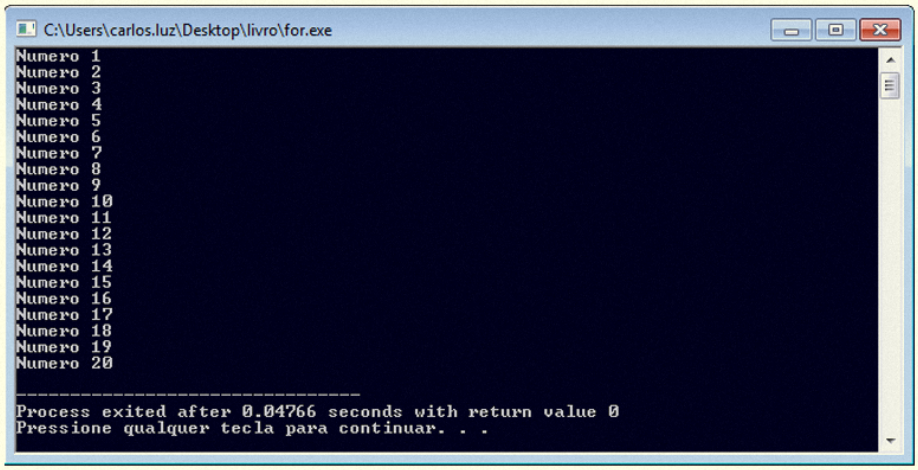
Prezado(a) aluno(a), veja que, como na instrução condicional, todo o código a ser executado pelo laço deve estar entre chaves, **{** ... **}**, pode-se efetuar o comando **for** sem elas quando tivermos apenas 1 linha do código a ser executado dentro do laço.

Para compreendermos melhor o laço de repetição **for**, observamos a Figura 1.24, na qual temos o código fonte de um programa em C que imprime na tela os números de 1 a 20.

```c
#include <stdio.h>

int i;

main(){

    for(i=1; i<=20; i++){

        printf("Numero %d \n", i);

    }

}
```

1FIGURA 24.59 - Exemplo do laço FOR FONTE: DEV C ++.

Veja o resultado do programa em C apresentado na Figura 1.24.



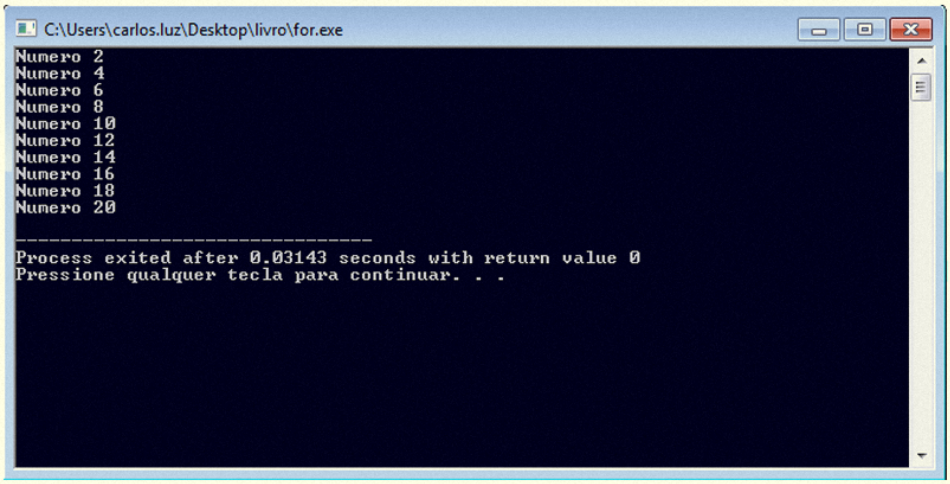
1FIGURA 25.59 - Resultado do programa de exemplo FOR FONTE: DEV C ++.

Podem-se utilizar várias funções e comandos em um laço de repetição, complementando o programa apresentado, vamos imprimir na tela agora somente os números que forem par:

```c
#include <stdio.h>

int i;

main(){

    for(i=1; i<=20; i++){

        if( i % 2 == 0) printf("Numero %d \n", i);

    }

}
```

1FIGURA 26.59 - Exemplo de aplicação do FOR junto com outros comandos FONTE: DEV C ++

O resultado do programa em C apresentado na Figura 1.26.



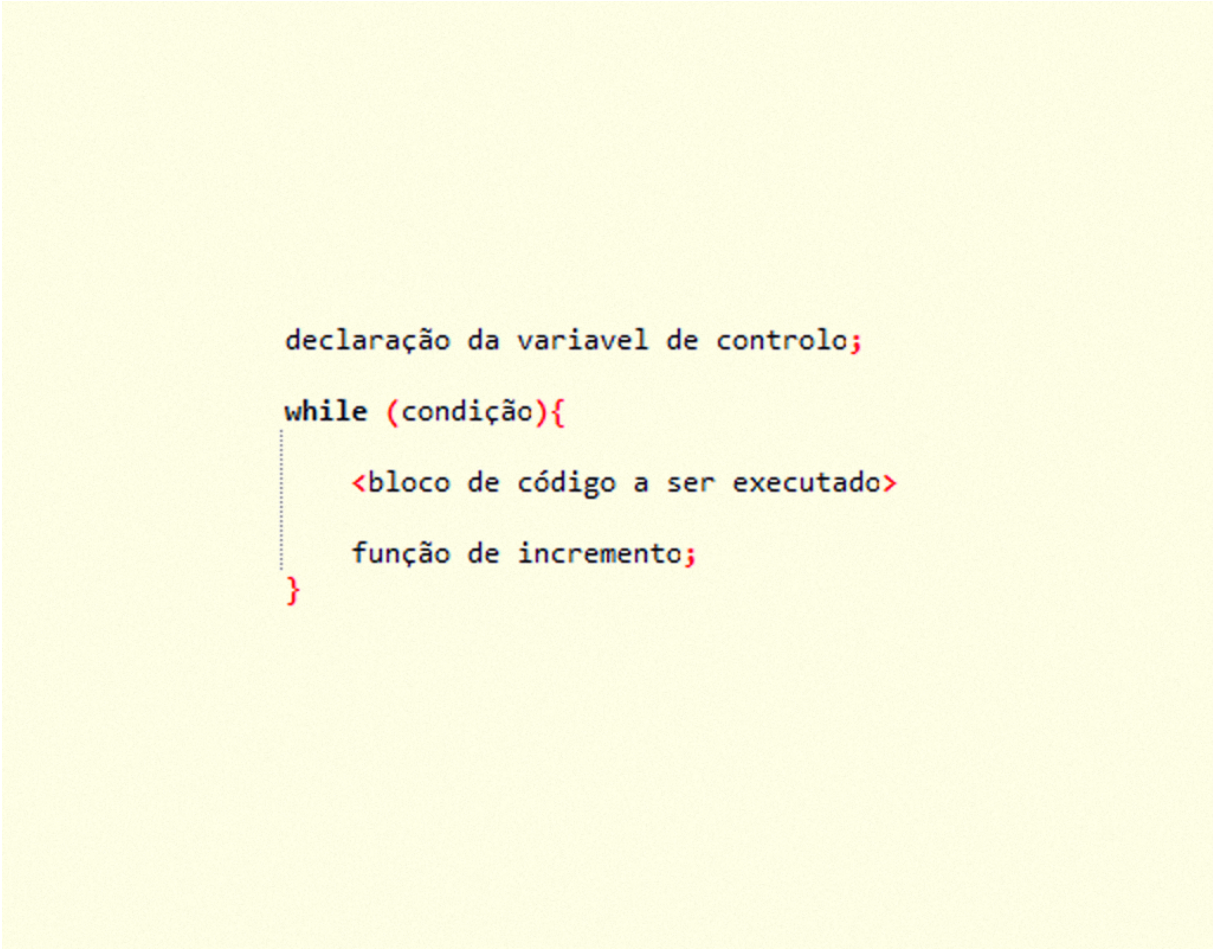
1FIGURA 27.59 - Resultado do exemplo FOR junto com os demais comandos FONTE: DEV C ++.

O que diferencia o laço **for** dos demais tipos é que determinamos a quantidade de iterações que ele deverá fazer no programa.

Estrutura de Repetição WHILE

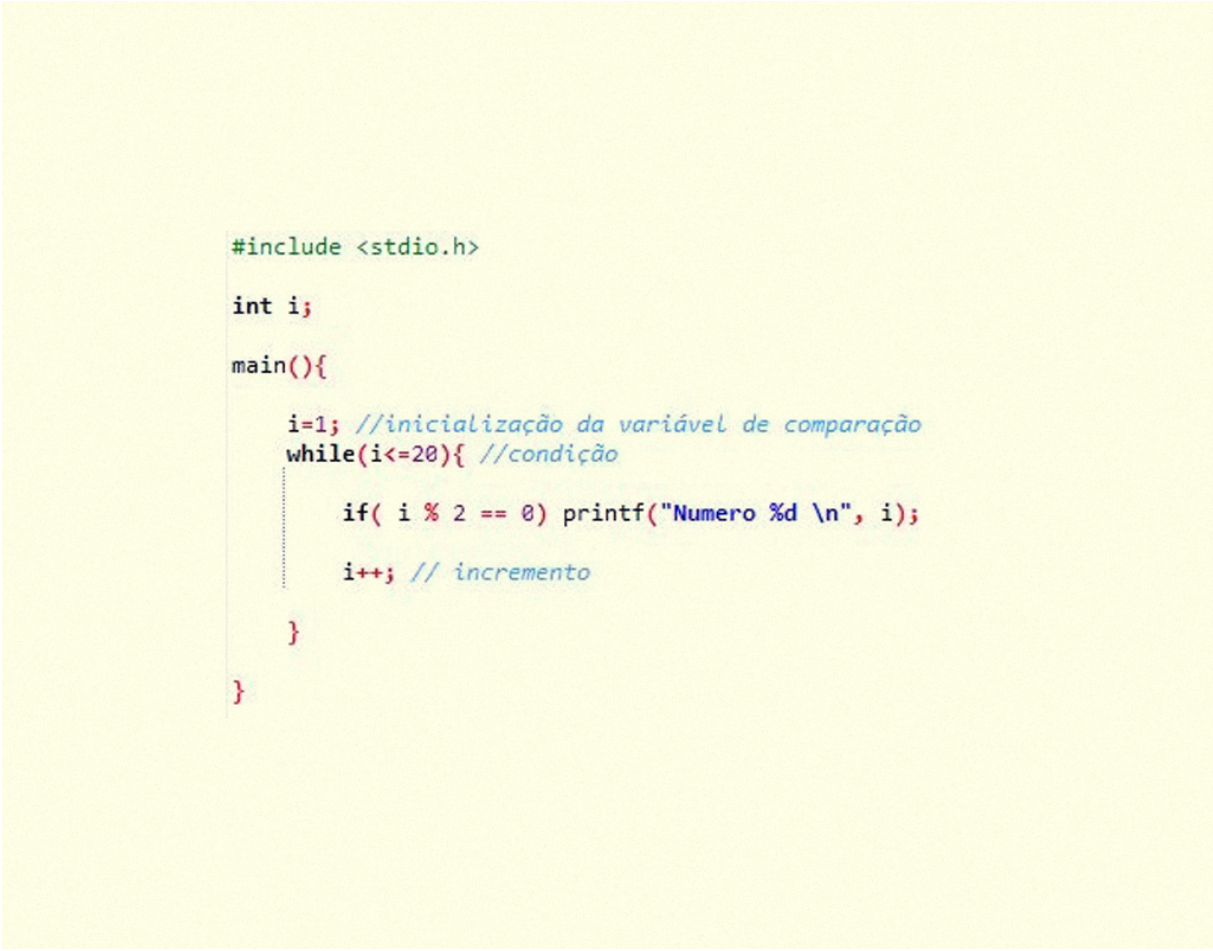
O comando **While** também é um laço de repetição igual ao **for** e se diferencia por ser um tipo de laço condicional, ele se baseia em uma condição para efetuar a repetição.

> O comando WHILE consiste na palavra-chave **while** seguida de uma expressão de teste entre parênteses. Se a expressão de teste for verdadeira, o corpo do laço é executado uma vez e a expressão de teste é avaliada novamente. Esse ciclo de teste e execução é repetido até que a expressão de teste se torne falsa (igual a zero), então o laço termina e o controle do programa passa para a linha seguinte ao laço .
>
> (MIZRAHI, 2008 p.73)



1FIGURA 28.59 - Composição textual do laço While FONTE: o autor.

A forma de iteração do **while** é bem parecida com a do laço **for**, o que os diferencia é que o laço **for** efetua as **x** iterações pré-definidas, já o **while**, ao ser executado, pode parar as iterações a qualquer momento se a condição inicial for atendida.

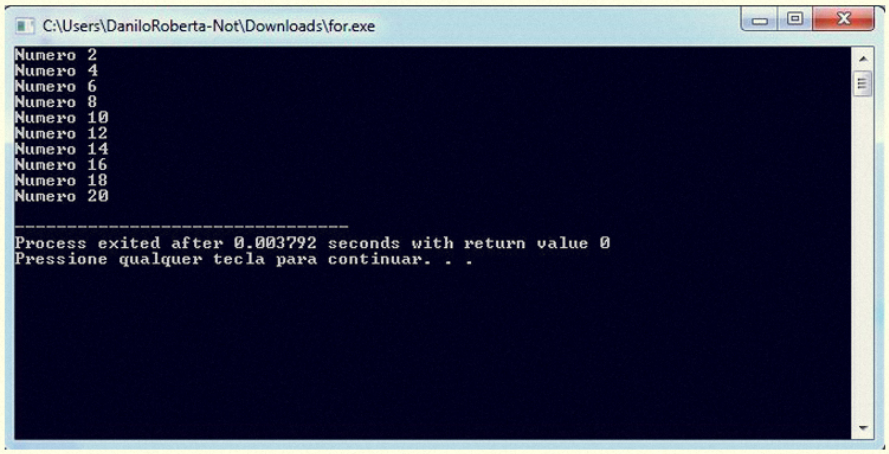


```c
#include <stdio.h>

int i;

main(){

    i=1; //inicialização da variável de comparação
    while(i<=20){ //condição

        if( i % 2 == 0) printf("Numero %d \n", i);

        i++; // incremento

    }

}
```

1FIGURA 29.59 - Exemplo de programa utilizando o laço While FONTE: DEV C ++.

O código fonte **while** apresentado efetua a mesma situação que o código **for**, o que o diferencia é que, antes de entrar no laço, é efetuada a consulta na condição, e ele valida a situação: **se** avariável i for **menor ou igual** a 20, o programa entra no laço **while**, a cada nova interação a função **i++;** incrementa um número na variável i.

O resultado do programa em C apresentado na Figura 1.29.



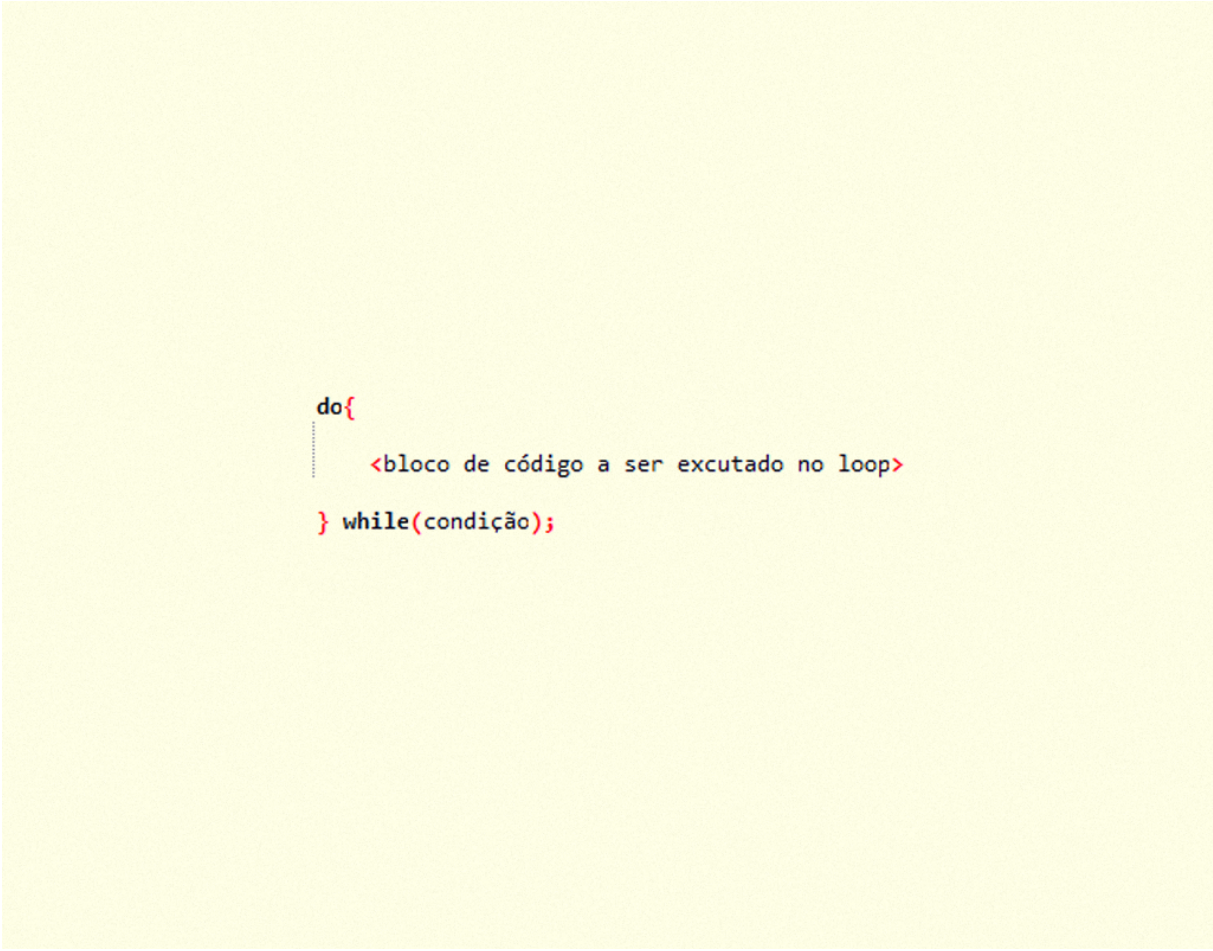
1FIGURA 30.59 - Resultado do programa de exemplo WHILE FONTE: DEV C ++.

Estrutura de Repetição DO WHILE

A estrutura de repetição **do... while** é semelhante à estrutura while. Na instrução while, a condição da continuação de loop é testada no início do loop, antes que seu corpo seja executado. A estrutura **do... while** testa a condição da continuação do loop depois que o corpo do loop é executado. Portanto, o corpo do loop será executado pelo menos uma vez (DEITEL, DEITEL 2011 p.93).

Pode-se compreender que o laço de repetição DO... WHILE testa a condição no final no laço, dessa forma, ele efetua o loop pelo menos uma vez, sendo encerrado somente se a condição de saída retornar como verdadeira na condição.

Veja a sintaxe do laço de repetição:



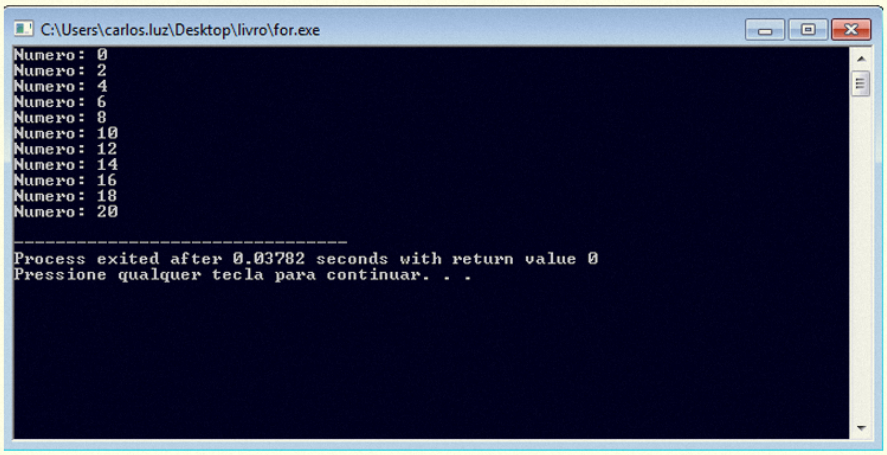
1FIGURA 31.59 - Composição textual do laço Do ... While FONTE: Dev C ++.

Ao apresentar a estrutura **while**, utilizamos um código fonte que efetuava um loop de 20 números e apresentava na tela apenas os números pares, podemos efetuar o mesmo laço de repetição com DO ... WHILE:

```c
#include <stdio.h>

int i;

main(){

    do{

        if( i % 2 == 0) printf("Numero: %d \n", i);

    }while(++i <=20);

}
```

1FIGURA 32.59 - Exemplo de de programa utilizando DO ... WHILE FONTE: DEV C ++.

Veja o resultado do programa em C apresentado na Figura 1.32:



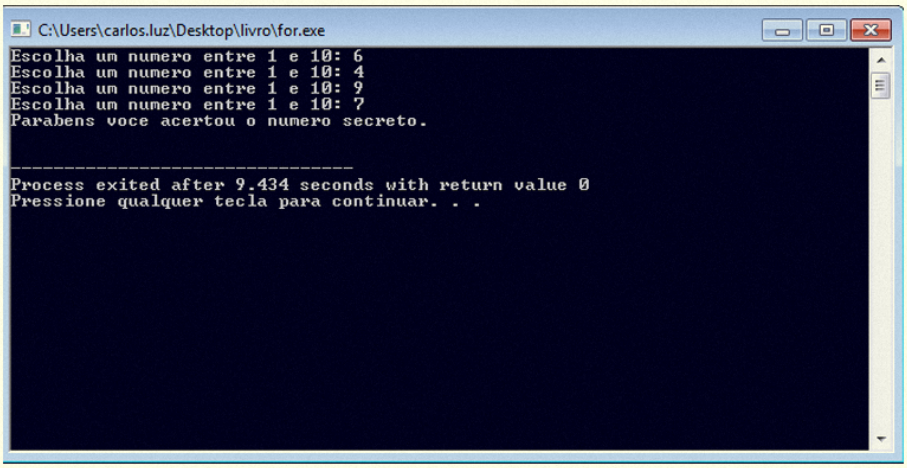
1FIGURA 33.59 - Resultado do programa utilizando DO ... WHILE FONTE: DEV C ++.

Podemos utilizar como condição de validação outros operadores além do >= ou <=, vejamos o código a seguir no qual o usuário deve descobrir qual o número secreto para poder sair do laço de repetição e concluir a execução do programa:

```c
#include <stdio.h>

int i;

main(){

    do{

        printf("Escolha um numero entre 1 e 10: ");
        scanf("%d", &i);

    }while( i != 7 );

    printf("Parabens voce acertou o numero secreto. \n \n");

}
```

1FIGURA 34.59 - Exemplo 2 utilizando DO ... WHILE FONTE: DEV C ++.

Resultado do programa em C apresentado na Figura 1.34.



1FIGURA 35.59 - Resultado do exemplo 2 utilizando DO ... WHILE FONTE: DEV C ++

Vetores, Matrizes e Estruturas

Utilizamos vetores, matrizes e estruturas (struct), quando é necessário termos muitas variáveis para o mesmo fim. Vamos pensar em um sistema que irá armazenar 10 nomes de pessoas, teríamos que ter 10 variáveis declaradas, dessa forma, ao utilizarmos um vetor, se declara apenas 1 variável.

¶ Reflita

Prezado(a) aluno(a), quando estamos falando sobre a quantidade de declarações de variáveis, essa quantidade pode interferir no bom desempenho do programa quando executado ou isso não importa? Com o código fonte menor, o seu entendimento fica confuso ou mais organizado?

Vetores

Alguns autores identificam um **vetor** como **array**, em tese, ambos podem ser considerados a mesma coisa e têm a mesma função.

Um array é um conjunto de espaços de memória que se caracterizam pelo fato de que todos têm o mesmo nome e o mesmo tipo de variável (CHAR, INT, FLOAT) (Adaptado de DEITEL, 2011 p.161).

Vetor também é conhecido como variável composta homogênea unidimensional. Isso quer dizer que se trata de um conjunto de variáveis de mesmo tipo, as quais possuem o mesmo identificador (nome) e são alocadas sequencialmente na memória. Como as variáveis têm o mesmo nome, o que as distingue é um índice que referencia sua localização dentro da estrutura (ASCENCIO, CAMPOS 2007 p.144).

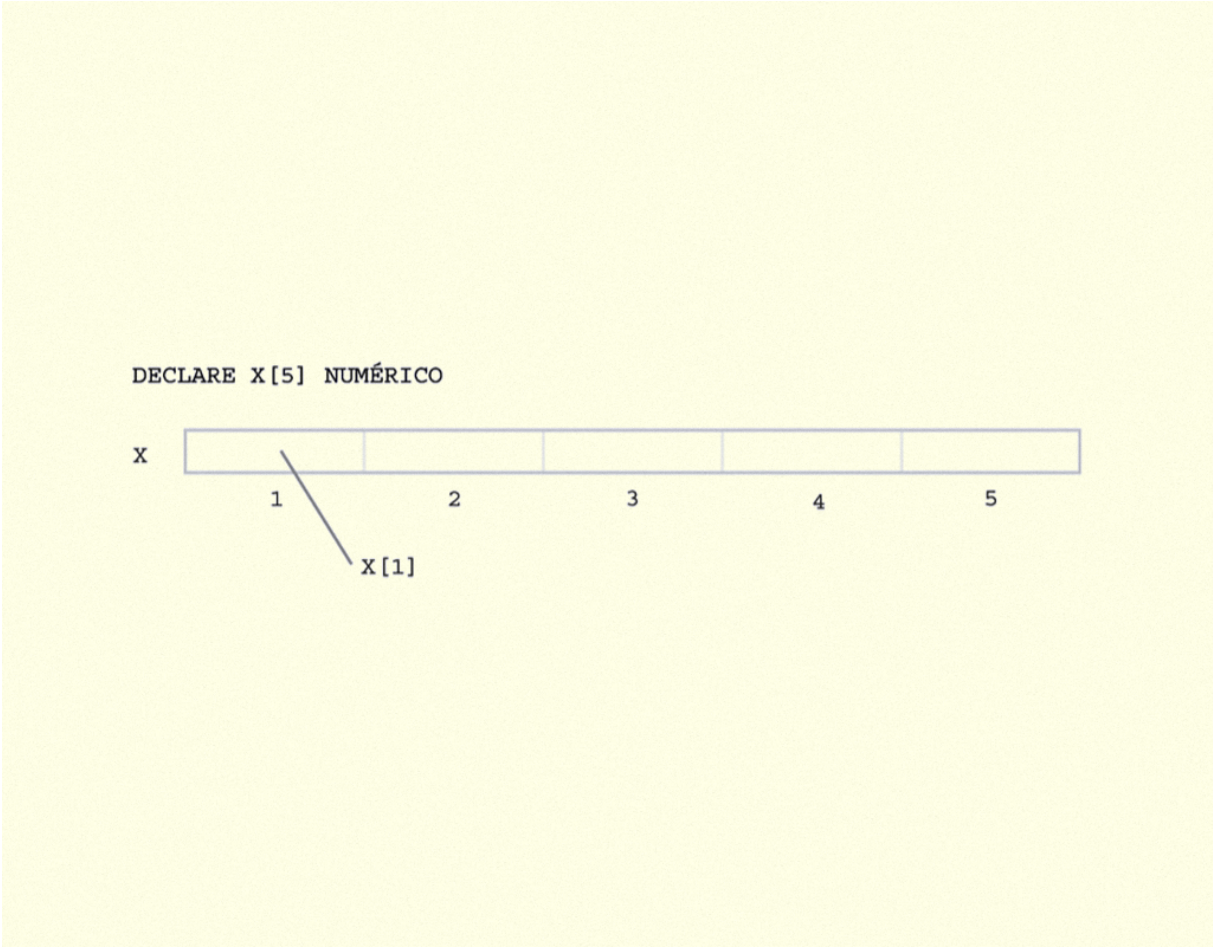
A entrada de dados nesse vetor só será aceito se ele for de igual tipo declarado. Exemplo: se o vetor for declarado com o tipo INT, só serão armazenados dados numéricos inteiros.

Veja a sintaxe da declaração de um vetor:

```
#include <stdio.h>

    // tipo nome variável [tamanho do vetor];

    int numero [5];

main(){

    <bloco de código>
}
```

1FIGURA 36.59 - Composição textual da declaração de um vetor FONTE: o autor.

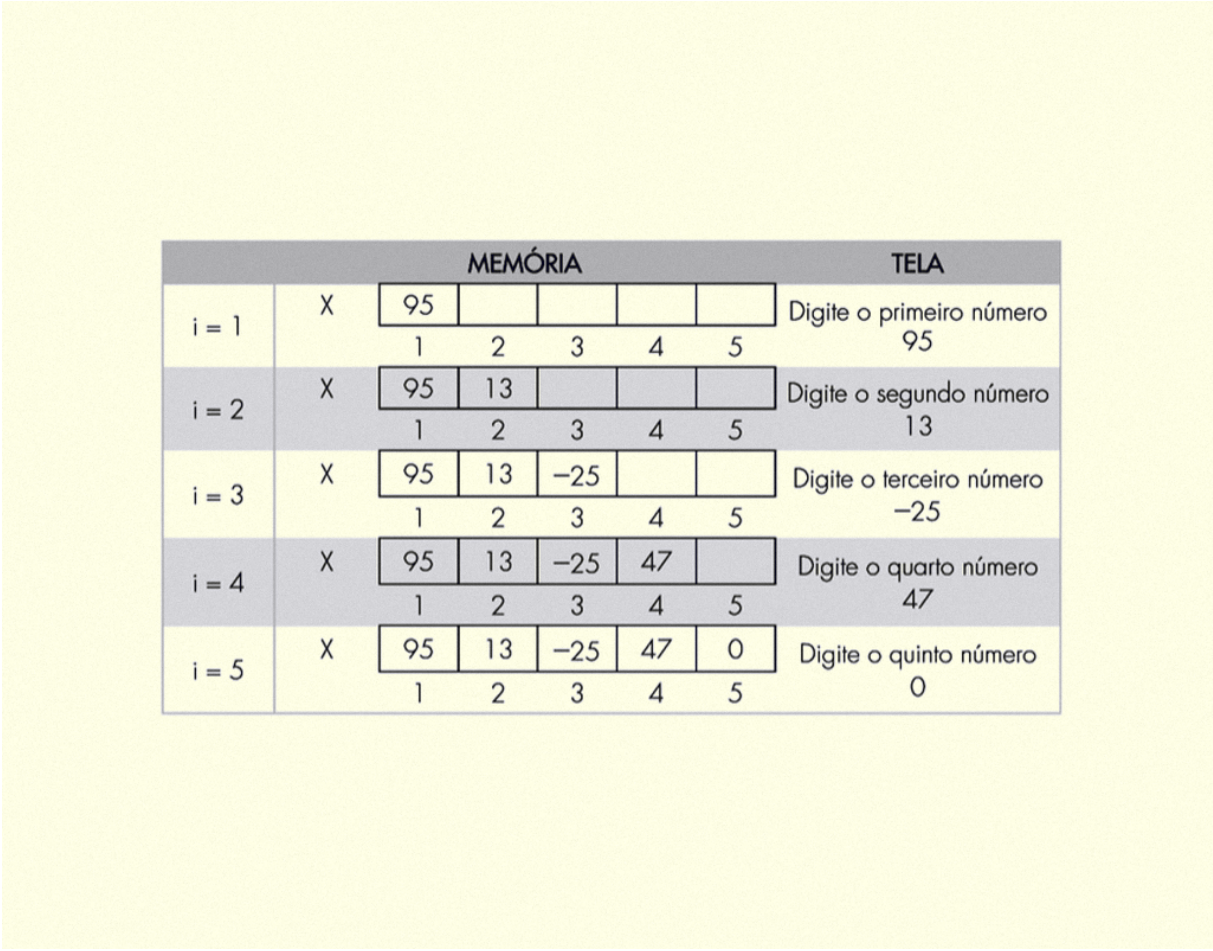
O código apresenta a declaração de um vetor com o nome **numero**, cujo tamanho será de **5**, do tipo **INT**. O tamanho do vetor aceita apenas números inteiros, pois esse valor representa o local que um dado será armazenado no vetor.



1FIGURA 37.59 - Representação de alocação de um dado no vetor FONTE: Ascencio e Campos (2007 p.144).

Para que possamos acessar algum local no vetor, devemos informar qual o local:

- x[1] = representa a 1° posição do vetor, é possível acessar o dado armazenado se referenciando a sua posição.
- x[4] = representa a 4° posição do vetor, é possível acessar o dado armazenado se referenciando a sua posição.
- x[45] = representa a 45° posição do vetor, é possível acessar o dado armazenado se referenciando a sua posição.



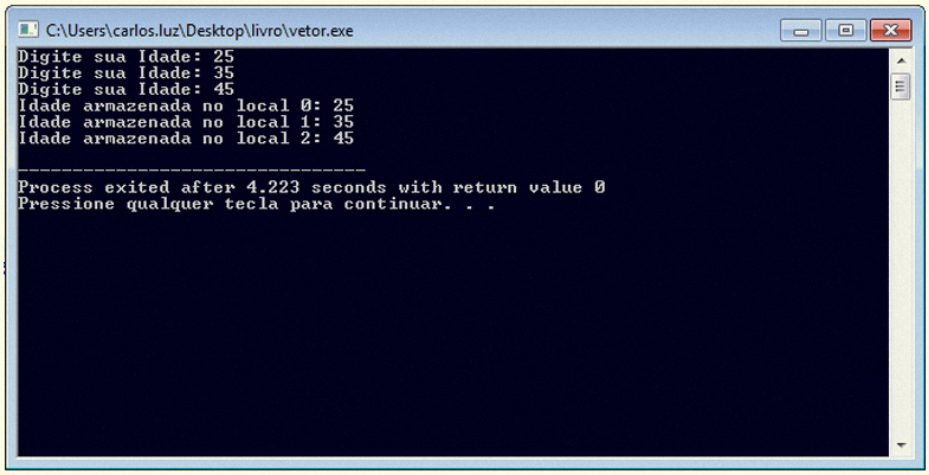
1FIGURA 38.59 - Exemplo de alocação de dados no vetor FONTE: Ascencio e Campos (2007 p.145).

Em um mesmo programa, podemos ter vários vetores de diversos tipos, normalmente as informações são inseridas no vetor através de um laço de repetição (FOR, WHILE ou DO ... WHILE), conforme visto no código a seguir.

```c
#include <stdio.h>

    int idade [3];
    int j, k;
main(){

    for(k=0; k<3; k++){

        printf("Digite sua Idade: ");
        scanf("%d", &idade[k]);
    }

    for(j=0; j<3; j++){

        printf("Idade armazenada no local %d: %d \n", j, idade[j]);
    }
}
```

1FIGURA 39.59 - Exemplo de programa utilizando variável com vetor FONTE: DEV C ++.

Resultado do programa em C apresentado na Figura 1.39.



1FIGURA 40.59 - Resultado do exemplo utilizando vetores FONTE: DEV C ++.

Através do laço de repetição **for**, foram inseridas 3 entradas de dados nas posições do vetor número, posteriormente, em outro laço de repetição **for**, foramapresentados os dados na tela.

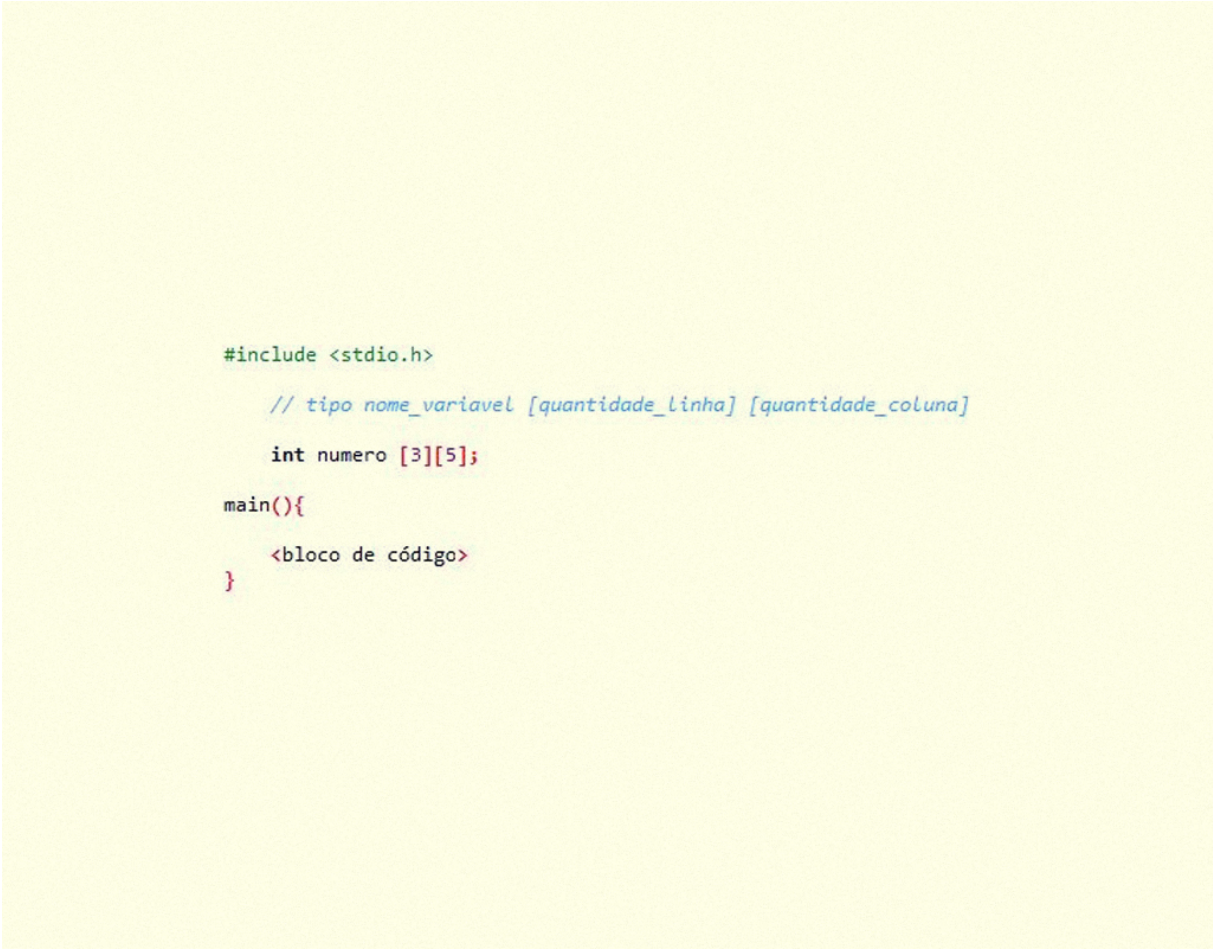
Um ponto muito importante que devemos observar é que todo e qualquer vetor ou matriz tem sua posição inicial como zero, por exemplo:

- ao declararmos um vetor de 3 posições **idade[3]**
- as posições respectivamente serão idade[0], idade[1] e idade [2]

Matrizes

Podemos considerar que uma matriz é um vetor melhorado, pois conforme visto anteriormente, os vetores guardam dados do mesmo tipo de variável em uma única linha, através de colunas. Uma matriz trabalha com linhas e colunas, sendo a primeira posição a quantidade de linha e a segunda, a de colunas, assim podemos multiplicar os espaços de armazenamento.

A sintaxe da declaração de uma matriz:



1FIGURA 41.59 - Composição textual da declaração de uma matriz FONTE: o autor.

Ao declararmos um vetor como sendo **idade[5]**, estamos informando que a variável irá armazenar 5 idades, cada uma em sua posição no vetor, ao efetuarmos o mesmo processo utilizando uma matriz **idade[5][3]**, será possível armazenar 5*3 = 15 registros, pois teremos 5 linhas e 3 colunas.

Para manipular os elementos dentro de uma matriz, são utilizadas as coordenadas de sua posição:

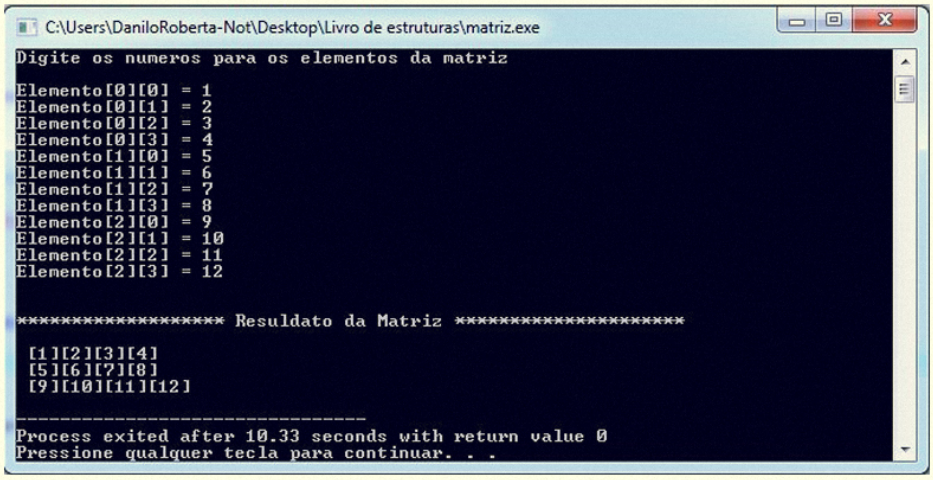
- idade[3][2] = será acessada a 3ª linha e buscados os dados contidos na 2ª coluna;
- numero[1][16] = será acessada a 1ª linha e buscados os dados contidos na 16ª coluna;
- idade[45][13] = será acessada a 45ª linha e buscados os dados contidos na 13ª coluna.

A forma de entrada de dados é muito parecida com a do vetor, mas dessa vez temos que utilizar dois laços de repetição, um que representa a linha, e outro, a coluna.

```c
#include <stdio.h>

main (){
  int numero[3][4];
  int m, p;

  printf ("\nDigite os numeros para os elementos da matriz\n\n");

  for ( m=0; m<3; m++ ){ // laço de repetição da linha

    for ( p=0; p<4; p++ ){ // laço de repetição da coluna

      printf ("Elemento[%d][%d] = ", m, p);
      scanf ("%d", &numero[ m ][ p ]);
    }
  }

   printf("\n\n****************** Resuldato da Matriz ****************** \n\n");

   for ( m=0; m<3; m++ ){


      printf (" [%d][%d][%d][%d] \n", numero[m-1][p], numero[m-1][p+1], numero[m-1][p+2], numero[m-1][p+3]);
    }

}
```

1FIGURA 42.59 - Exemplo de programa utilizando uma matriz FONTE: DEV C ++.

Resultado do programa em C apresentado na Figura 1.42.



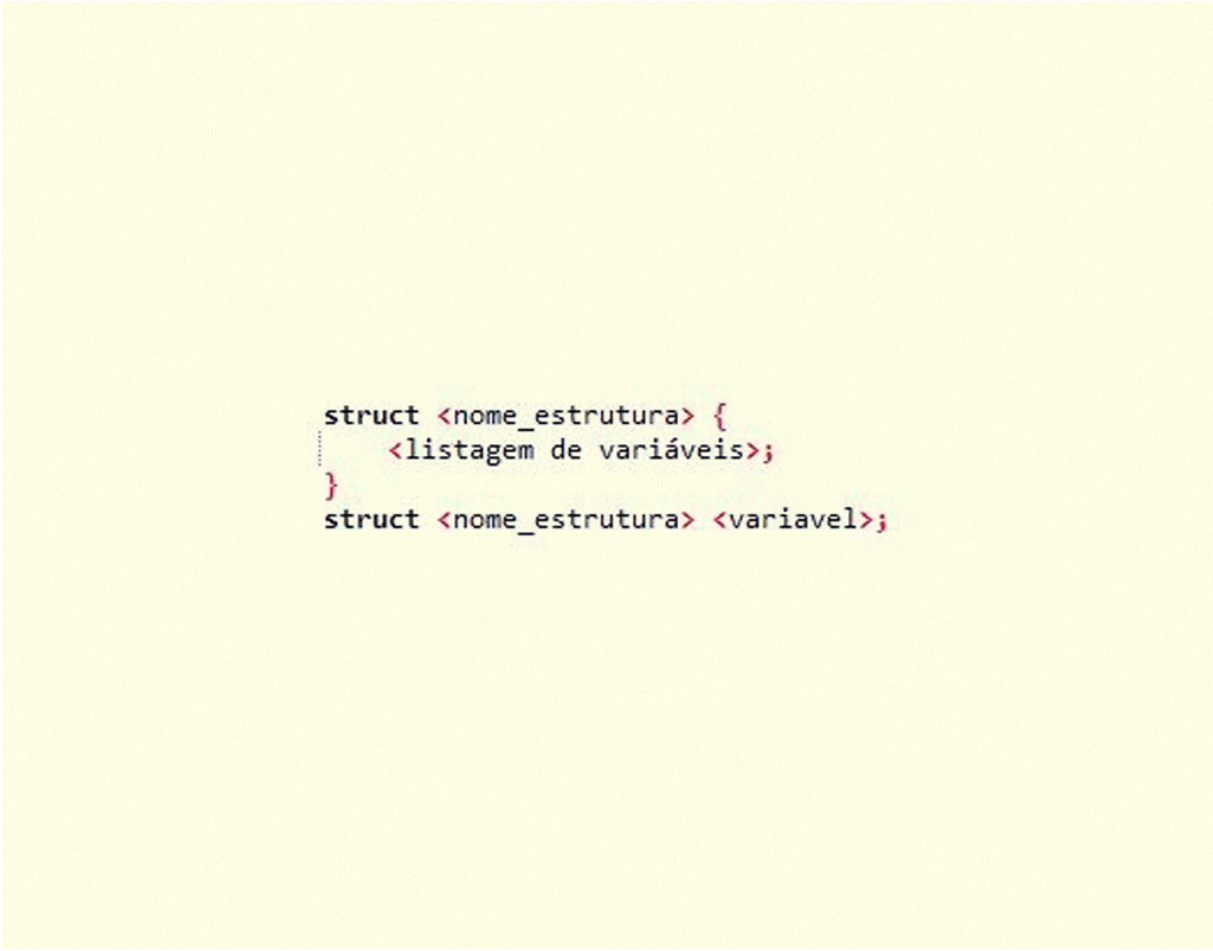
1FIGURA 43.59 - Resultado do programa utilizando uma Matriz FONTE: DEV C ++

Estruturas

Os vetores e matrizes são utilizados para o agrupamento de variáveis que sejam do mesmo tipo (INT, FLOAT, DOUBLE, CHAR). Já o comando **struct** efetua o agrupamento de diferentes tipos de variáveis.

Por meio da palavra-chave **struct**, definimos um novo tipo de dado. Definir um tipo de dado significa informar ao compilador seu nome, tamanho em bytes e forma como deve ser armazenado e recuperado da memória. Após ter sido definido, o novo tipo existe e pode ser utilizado para criar variáveis de modo similar a qualquer tipo simples (MIZRAHI, 2008 p.223).

Observe a sintaxe da declaração de uma matriz:



1FIGURA 44.59 - Composição textual da declaração de uma struct (estrutura) FONTE: o autor.

Em uma **struct**,é possível agrupar uma lista de n variáveis de diversos tipos, as quais são declaradas entre as chaves **{ ... }**, da mesma forma como se estivessem sendo declaradas as variáveis fora das chaves.

```c
#include <stdio.h>

struct dados_pessoa{
    char nome[25];
    int idade;
    float altura;
};

struct dados_pessoa pessoa;

main(){

    printf("Digite seu nome:");
    scanf("%s", &pessoa.nome);

    printf("Digite sua idade:");
    scanf("%d", &pessoa.idade);

    printf("Digite sua altura:");
    scanf("%f", &pessoa.altura);

    printf("\nPessoa:%s Idade:%d Altura:%0.2f", pessoa.nome, pessoa.idade, pessoa.altura);
}
```

1FIGURA 45.59 - Exemplo de programa utilizando Struct FONTE: DEV C ++.

Resultado do programa em C apresentado na Figura 1.45.

1FIGURA 46.59 - Resultado do Exemplo utilizando Struct FONTE: DEV C ++

Funções e Ponteiros

Em certos pontos de um programa, temos que repetir um bloco de código por várias vezes, essa repetição deixará nosso código fonte muito extenso podendo causar lentidão quando executado.

Podemos utilizar funções para evitar que um trecho seja repetido por várias vezes de modo desnecessário, havendo também um reaproveitamento do código já desenvolvido.

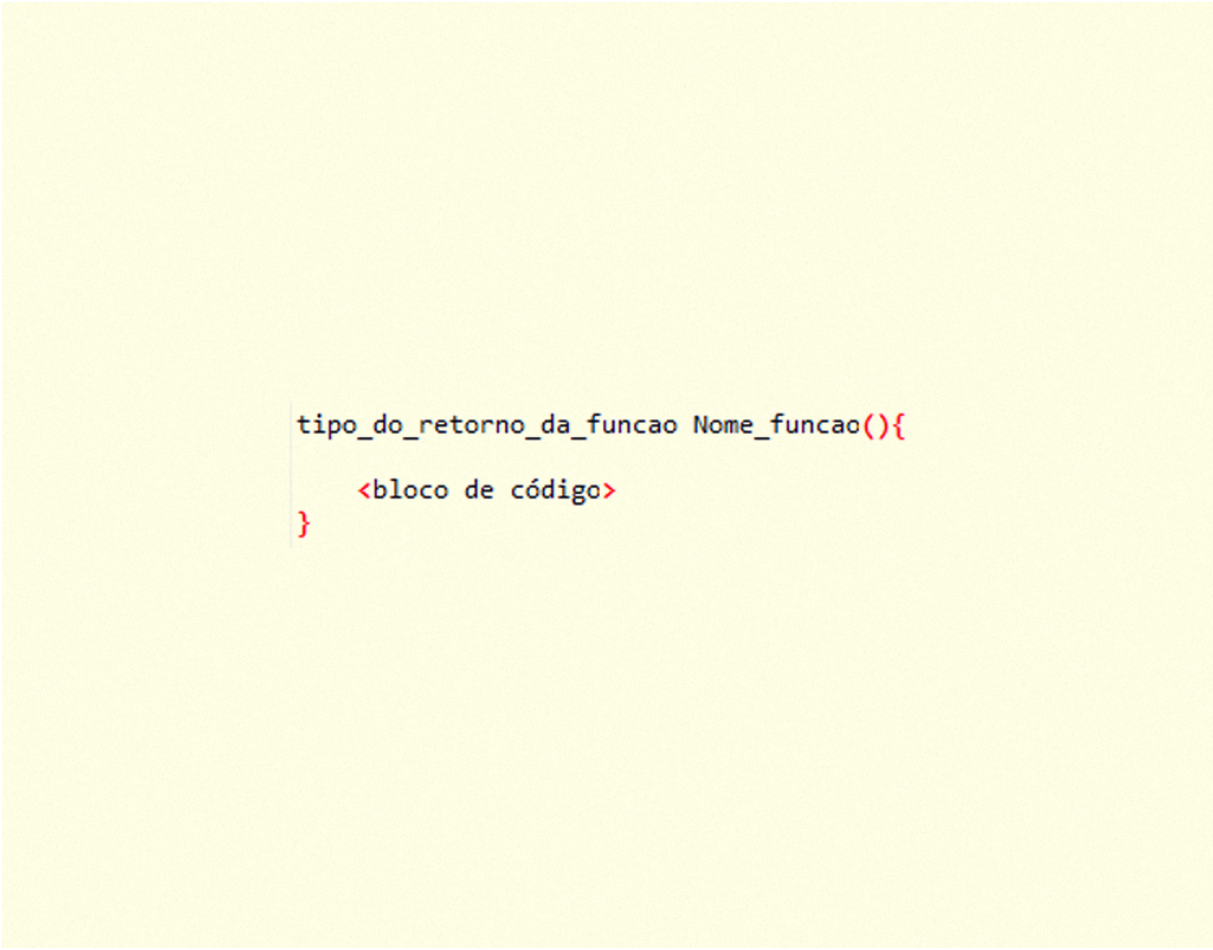
Pensando no aproveitamento de código, podemos utilizar os ponteiros, os quais podem assumir o armazenamento de outras variáveis, posteriormente efetuando o apontamento do local.

Funções

Os principais benefícios de se utilizarem funções seriam: uma melhor organização e reaproveitamento do código, consequentemente, um melhor entendimento.

As funções são blocos de construção em C, em que ocorrem todas as atividades do programa. Assim que uma função tenha sido escrita e depurada, poderá ser reutilizada quantas vezes for necessário (SCHILDT, 1986 p.78).

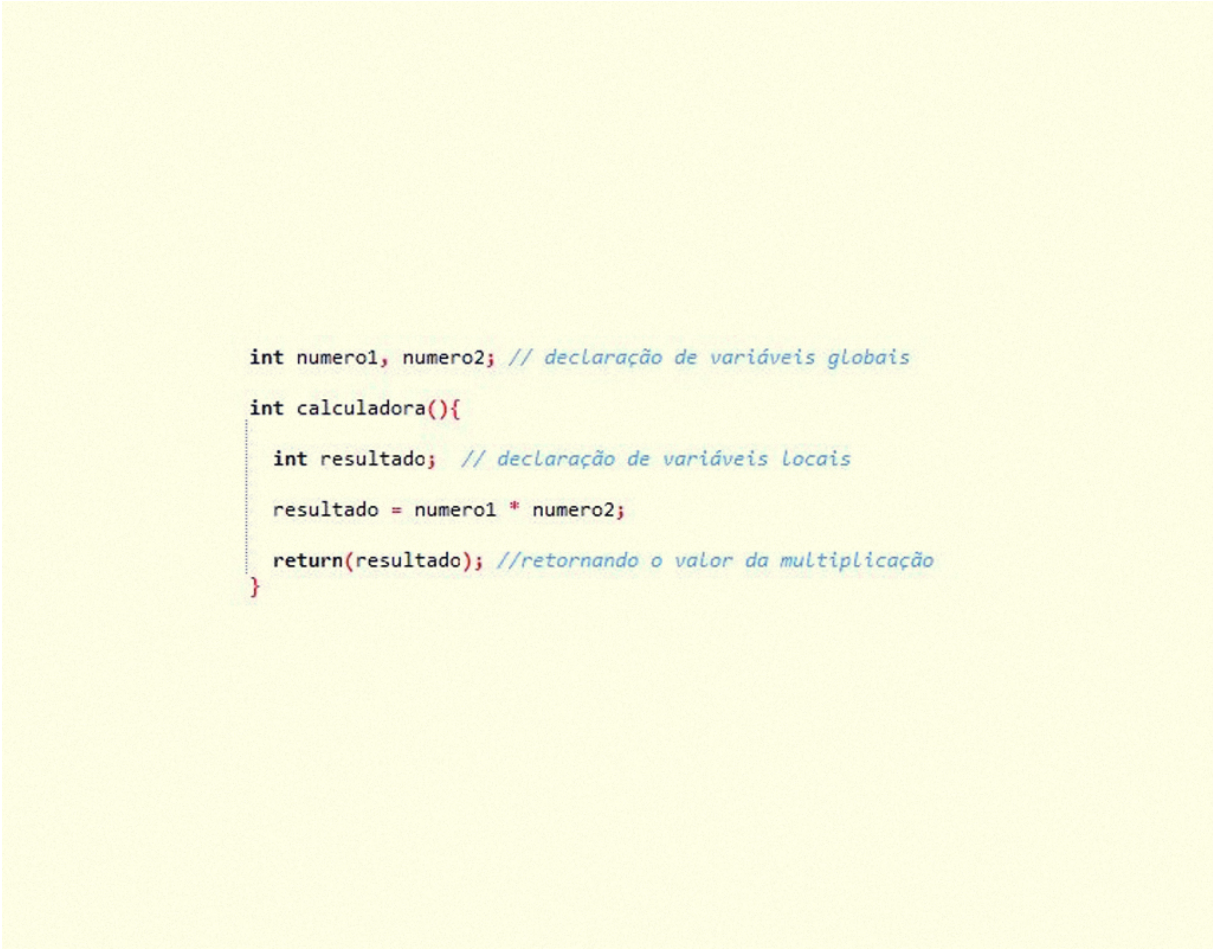
Veja a sintaxe da declaração de uma função:



1FIGURA 47.59 - Composição textual da declaração de função FONTE: o autor.

Podem-se declarar as variáveis que serão utilizadas na função de três formas: **globais**, **locais** e **parâmetros formais**.

- Variáveis globais: são declaradas fora da função, como as demais, Albano e Albano (2010 p.132) complementam dizendo que "***a mesma pode ser utilizada ou acessada por qualquer função ou bloco de comando.***"
- **Variáveis locais**: são declaradas dentro da função a ser utilizada, Albano e Albano (2010 p.132) afirmam que "***as variáveis locais pertencem apenas à função onde foram declaradas, não podendo, portanto, ser acessadas por meio de outra função de forma direta***".
- **Parâmetros formais**: são declarados na passagem de dados para a função, veremos com mais detalhe no próximo tópico.



```c
int numero1, numero2; // declaração de variáveis globais

int calculadora(){

  int resultado;  // declaração de variáveis locais

  resultado = numero1 * numero2;

  return(resultado); //retornando o valor da multiplicação
}
```

1FIGURA 48.59 - Exemplo da criação de uma função. FONTE: Dev C ++.

Vamos analisar os códigos a seguir, sendo um desenvolvido de modo convencional e outro com a aplicação de funções:

```c
#include<stdio.h>

int valor1, valor2, total1, total2, total3;

main( ){

    printf(" ******* 1 soma ******* ");

    printf("\n\nDigite o primeiro numero :");
    scanf ("%d", &valor1);
    printf("Digite o segundo numero :");
    scanf ("%d", &valor2);
    total1 = valor1+valor2;
    printf ("A soma e %d", total1);

    printf("\n\n  ******* 2 soma ******* ");

    printf("\n\nDigite o primeiro numero :");
    scanf ("%d", &valor1);
    printf("Digite o segundo numero :");
    scanf ("%d", &valor2);
    total2 = valor1+valor2;
    printf ("A soma e %d", total2);

    printf("\n\n  ******* 3 soma ******* ");

    printf("\n\nDigite o primeiro numero :");
    scanf ("%d", &valor1);
    printf("Digite o segundo numero :");
    scanf ("%d", &valor2);
    total3 = valor1+valor2;
    printf ("A soma e %d", total3);

    printf ("\n\nSoma total %d", (total1+total2+total3));

}
```

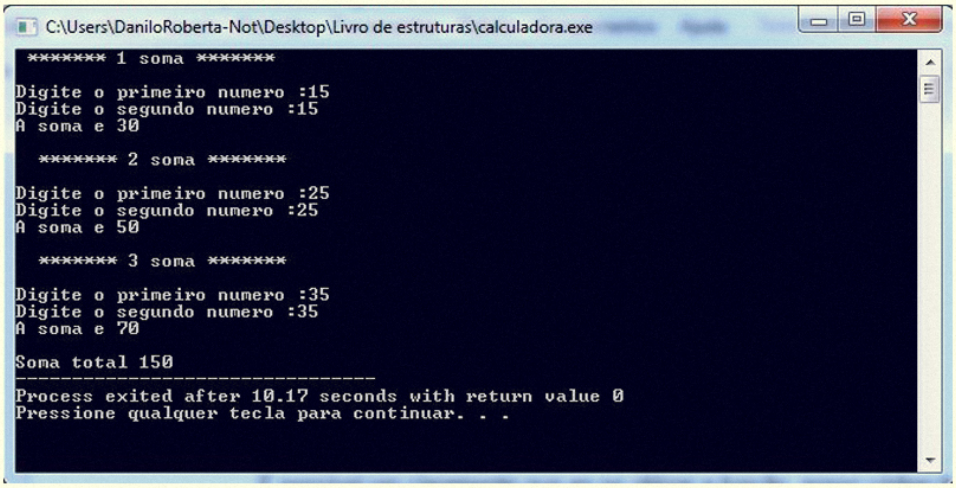
1FIGURA 49.59 - Exemplo de programa sem utilizar funções FONTE: DEV C ++.

```c
#include<stdio.h>

int valor1, valor2, total1, total2, total3;

int soma(){
        printf("\n\nDigite o primeiro numero :");
        scanf ("%d", &valor1);
        printf("Digite o segundo numero :");
        scanf ("%d", &valor2);
        return(valor1+valor2); //retornando o valor dos valores
    }

main( ){

    printf(" ******* 1 soma ******* ");
    total1 = soma();
    printf ("A soma e %d", total1);

    printf("\n\n  ******* 2 soma ******* ");
    total2 = soma();
    printf ("A soma e %d", total2);

    printf("\n\n  ******* 3 soma ******* ");
    total3 = soma();
    printf ("A soma e %d", total3);

    printf ("\n\nSoma total %d", (total1+total2+total3));

}
```

1FIGURA 50.59 - Código com aplicação de funções FONTE: DEV C ++.

É possível ver claramente que, ao se utilizar a função, nosso código fonte ficou menor, podendo agora efetuar a chamada da função **soma()** em qualquer parte do código fonte.

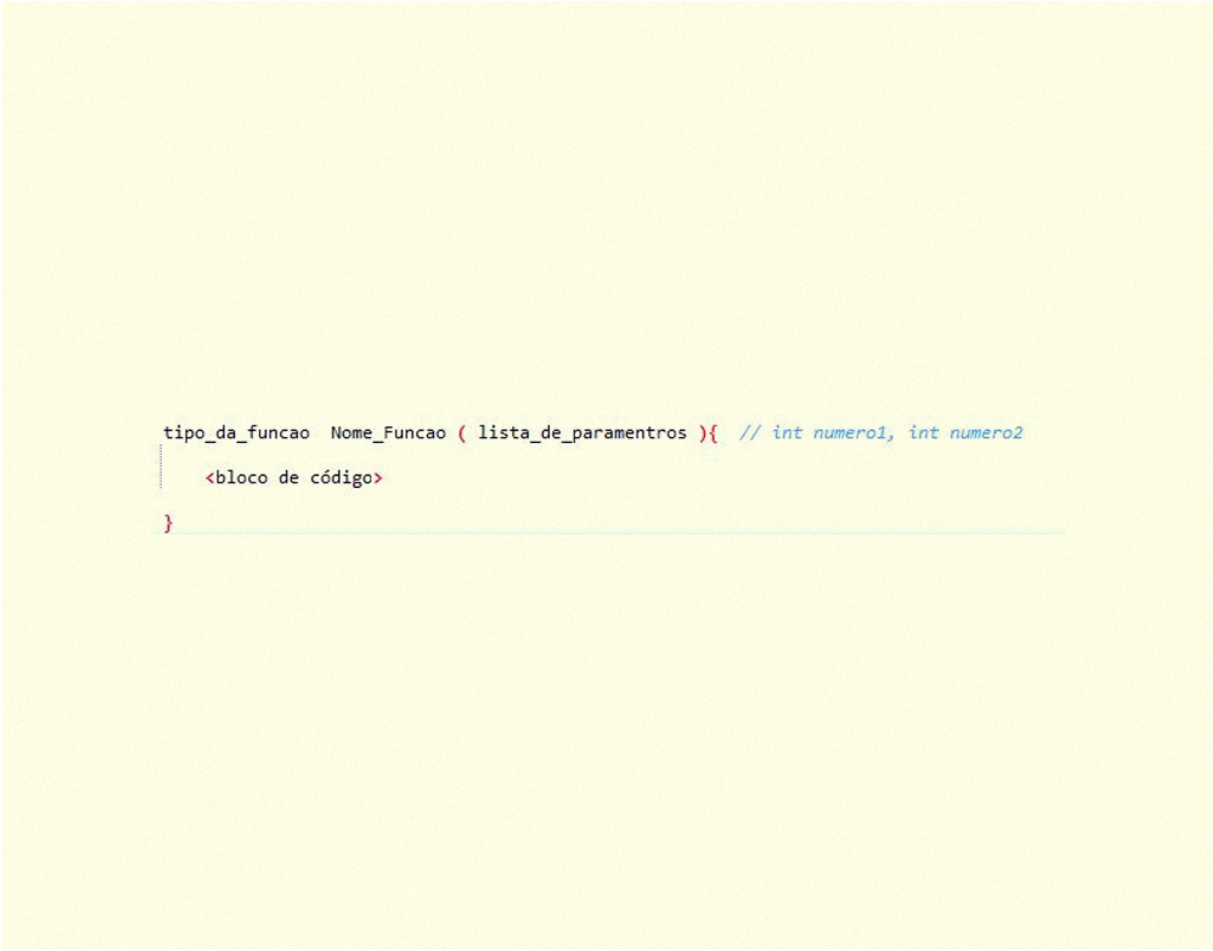
Resultado do programa em C apresentado na Figura 1.50.



1FIGURA 51.59 - Resultado do programa utilizando a função FONTE: DEV C ++.

Informando parâmetros de função

Ao utilizarmos funções, em alguns casos, temos que informar alguns dados para a manipulação dessas, pode-se efetuar esse comando por meios das variáveis de **parâmetros formais**. A sintaxe da função sofre apenas algumas modificações, pois é inserida uma lista de parâmetros esperada para ser utilizada dentro da função.



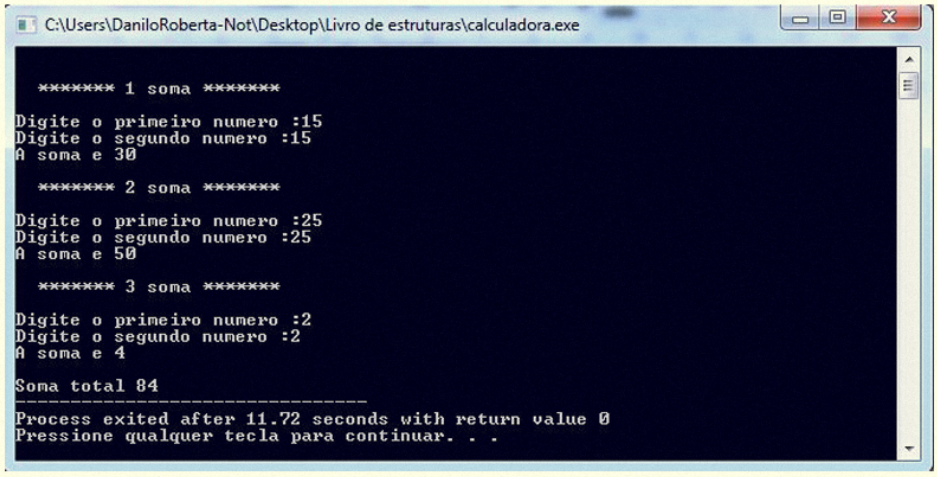
1FIGURA 52.59 - Composição textual da passagem de parâmetros da função FONTE: o autor.

Na passagem de parâmetros, os dados das variáveis são armazenados em novas variáveis, sendo respeitada a respectiva ordem de parâmetros. Temos então dois tipos: os parâmetros reais, que são dados obtidos na entrada pelo usuário, e os formais, que são os parâmetros declarados na função.

```c
#include<stdio.h>

int valor1, valor2, total[3];

    int soma(int N1, int N2){ // parâmetros formais
        int total_soma;
        total_soma = N1+ N2;
        return(total_soma); //retornando o valor dos valores
    }

main( ){

    for(int h=0; h<3; h++){
        printf("\n\n  ******* %d soma ******* ", h+1);
        printf("\n\nDigite o primeiro numero :");
        scanf ("%d", &valor1);
        printf("Digite o segundo numero :");
        scanf ("%d", &valor2);

        total[h] = soma(valor1, valor2); //valores sendo informados para a função
        printf ("A soma e %d", total[h]);

    }
    printf ("\n\nSoma total %d", (total[0]+total[1]+total[2]));

}
```

1FIGURA 53.59 - Exemplo de programa utilizando passagem de parâmetros para a função FONTE: DEV C ++.

Resultado do programa em C apresentado na Figura 1.53.



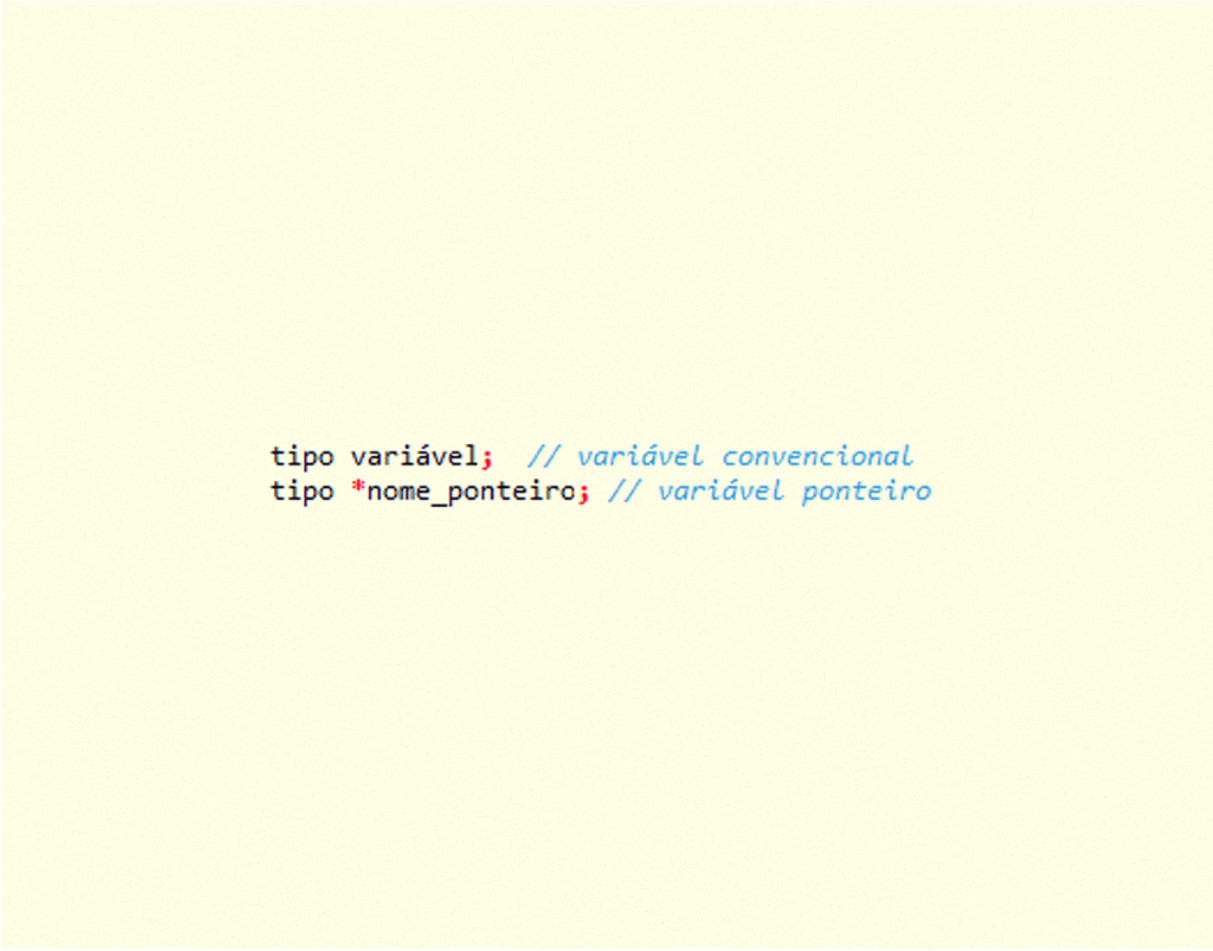
1FIGURA 54.59 - Resultado do programa utilizando passagem de parâmetros FONTE: DEV C ++.

Ponteiros

Como já apresentado, ao declararmos uma variável, ela se reserva um espaço na memória para armazenamento dos dados, cada espaço possui um endereço lógico, um número hexadecimal e, por meio desse endereço, é possível recuperar as informações armazenadas. Calma, prezado(a) aluno(a), não é necessário saber qual é o endereço lógico, pois quem efetua essa manipulação é o próprio SO (Sistema Operacional) e o programa em execução.

Basicamente, um ponteiro é uma variável que armazena um endereço de memória, isto é, o ponteiro é um tipo de variável capaz de atribuir somente os endereços de outra variável ao seu conteúdo, o qual é bem diferente das variáveis comuns com as quais estávamos trabalhando até o momento (ALBANO, ALBANO 2010 p.156).

Um ponteiro é considerado uma variável, o modo de declará-lo é bem parecido com as demais, a única diferença é que antes do nome do ponteiro é inserido o caractere asterisco *, sintaxe de declaração de um ponteiro:



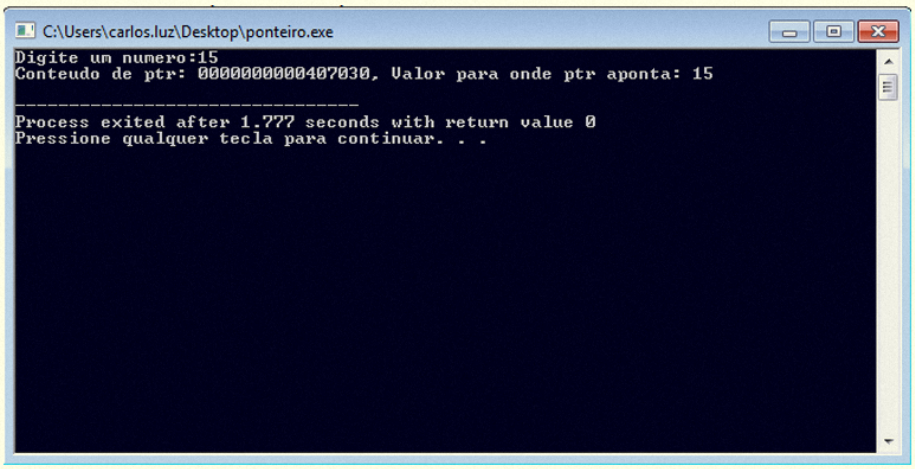
1FIGURA 55.59 - Composição textual da declaração de um ponteiro FONTE: o autor.

Pode-se utilizar um ponteiro com variável de diversos tipos como INT, FLOAT, CHAR e também STRUCT, vejamos a seguir um código no qual se utiliza um ponteiro do tipo inteiro:

```c
#include <stdio.h>

int numero;
int *ptr;   // declara um ponteiro para um inteiro

main (){

   printf ("Digite um numero:");
   scanf("%d", &numero);

   ptr = &numero;

   printf("Conteudo de ptr: %p, Valor para onde ptr aponta: %d \n", ptr, *ptr);

}
```

1FIGURA 56.59 - Exemplo de programa utilizando ponteiro FONTE: DEV C ++.

Resultado do programa em C apresentado na Figura 1.56.



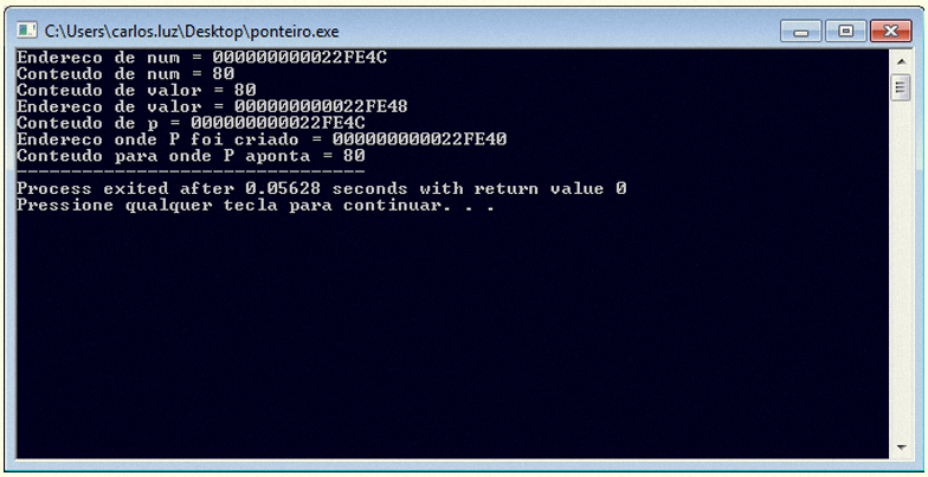
1FIGURA 57.59 - Resultado do programa utilizando ponteiro FONTE: DEV C ++.

Vejamos agora outro exemplo de ponteiros.

```c
#include <stdio.h>

main (){
   int num, valor, *p;
   num=80;
   p=&num;      // p armazena o endereco de num
   valor=*p;    // variável valor armazena o conteúdo de num
   printf ("Endereco de num = %p" , &num);
   printf ("\nConteudo de num = %i" , num);
   printf ("\nConteudo de valor = %i" , valor);
   printf ("\nEndereco de valor = %p" , &valor);
   printf ("\nConteudo de p = %p" , p);
   printf ("\nEndereco onde P foi criado = %p" , &p);
   printf ("\nConteudo para onde P aponta = %i" , *p);
}
```

1FIGURA 58.59 - Exemplo 2 de programa utilizando ponteiros FONTE: DEV C ++.

Resultado do programa em C apresentado na Figura 1.58.



1FIGURA 59.59 - Resultado do exemplo 2, utilizando ponteiros FONTE: DEV C ++.

Indicação de leitura

Nome do livro: Programação em Linguagem C

Editora: Ciência Moderna

Autor: ALBANO, Ricardo Sonaglio; ALBANO, Silvie Guedes.

ISBN: 978-857393-949-1

Sinopse: A linguagem C é utilizada na área de programação. O livro "Programação em linguagem C" oferece mais de 200 códigos-fontes, distribuídos entre exemplos e exercícios de fixação. É indicado para alunos de cursos de graduação, técnicos ou cursos livres. Além disso, os autodidatas poderão utilizar este livro, já que ele abrange, de forma sequencial, a fase introdutória da linguagem de programação C até a sua fase intermediária. Essa obra contém vários exercícios executados passo a passo que permitem que o leitor possa acompanhar o desenvolvimento de maneira útil e eficaz. Dessa forma, o próprio leitor poderá implementar cada exercício à medida que vai lendo o livro. A obra apresenta-se

estruturada de forma que, ao final de cada capítulo, sejam apresentados exercícios de revisão abrangendo cada conteúdo estudado, com o objetivo de avaliar e consolidar os conhecimentos adquiridos. Salientando que todos os exercícios possuem resolução contida no final do livro.

UNIDADE II

Estrutura de Dados

Carlos Danilo Luz

Caro(a) aluno(a)! Seja bem-vindo(a)! Nesta unidade, aprenderemos os conceitos iniciais sistemas números e alocação de dados na memória. Quando estamos executando qualquer programa, os dados são armazenados temporariamente na memória para que posteriormente seja possível manipulá-los. Serão apresentadas a você as árvores binárias, uma forma de organização dos dados armazenados na memória, nesse sentido, veremos alguns dos tipos de árvores disponíveis. Estudaremos também sobre listas lineares encadeadas, duplamente encadeadas, circulares e ordenadas, por fim, veremos os conceitos de fila e pilha atribuídos a listas lineares. Ao final de cada tópico, será vista uma implementação dos conceitos em linguagem C.

Sistemas numéricos e alocação de memória

Conforme visto na unidade anterior, os dados de uma variável são armazenados em um espaço na memória secundária do computador no momento de execução do programa. Para o computador, tudo são números, letras, caracteres especiais, a CPU efetua a conversão das informações e apresenta para o usuário.

Vejamos uma frase composta por números na imagem a seguir:

0100	0101	0110	1001	0111	0011	0010	0000	0110
0111	0001	0111	0101	0110	1001	0010	0000	0110
0110	1100	0110	0111	0111	0101	0110	1101	0110
0111	0011	0010	0000	0111	0000	0110	1101	0110
0110	1101	0111	0110	0111	0010	0110	0001	0111
0010	1110							

2QUADRO 1.11 - undefined

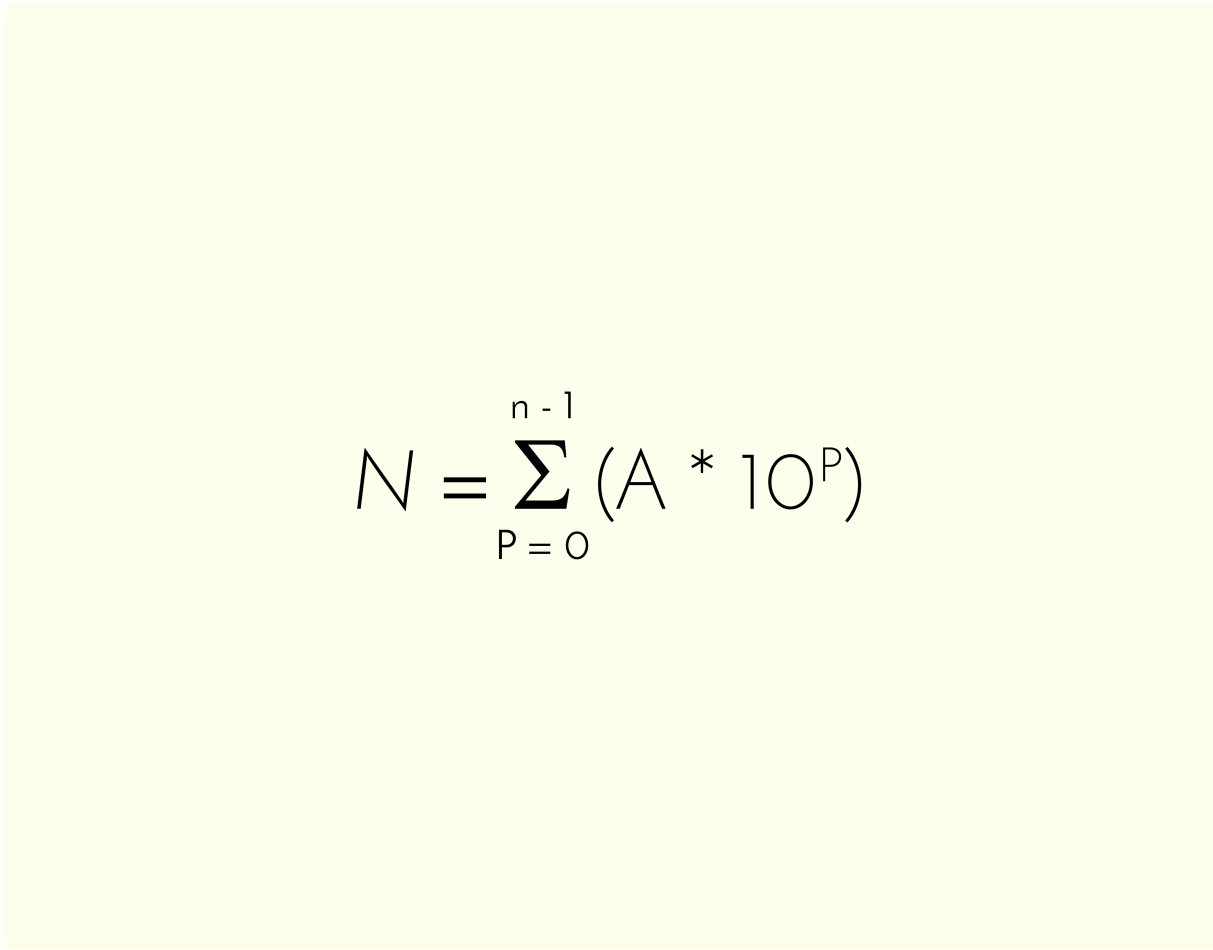
Frase escrita com números binários que diz: "Eis algumas palavras"

adaptado de Norton (1943, p. 103).

Sistema Decimal

Sistema Decimal, ou Sistema Base 10, é o primeiro tipo de sistema que aprendemos para realizar operações algébricas básicas: subtração, adição, divisão e multiplicação. Os números do sistema decimal podem assumir apenas 10 valores, sendo de 0 a 9, por isso seu nome.

Ao trabalharmos com o sistema decimal, a composição dos números se dá pela multiplicação de cada número por uma potência de base 10, cada expoente e informado com base na sua posição, sendo efetuada a leitura da direita para a esquerda iniciando por zero.



$$N = \sum_{P=0}^{n-1} (A * 10^{P})$$

2FIGURA 1.18 - Fórmula do sistema decimal FONTE: o autor.

Vejamos como exemplo o número em base 10 (decimal) : N (12345)10

O valor de N é o somatório da multiplicação dos números elevados na potência de base 10, conforme informado, a leitura é da direita para esquerda iniciando por 0.

N= 1*104 + 2*10^{3} + 3*10^{2} + 4*10^{1} + 5*100

N = 1*104 = 1* 10000 = 10000

2*10^{3} = 2 * 1000 = 2000

3*10^{2} = 3 * 100 = 300

4*10^{1} = 4 * 10 = 40

5*100 = 5 * 1 = 5

Total = 12345

Podemos compreender de forma simples e clara que 178910 é equivalente a 1789. É importante saber com é realizado esse cálculo, pois ele é utilizado em alguns casos para efetuarmos conversões de outros tipos de sistemas ou números para a base 10.

Sistema Binário

O sistema binário é o mais conhecido no mundo da tecnologia da informação e em diversas áreas; 0 e 1, ou ligado e desligado, entre outros, são algumas representações desse tipo de sistema. Mesmo que não estejamos vendo imagens, letras, músicas em MP3, vídeos em MP4, entre outros itens, são formados por um conjunto de zeros e uns.

Ao compararmos o sistema decimal com o binário, percebemos que antes tínhamos dez valores a serem utilizados (0 a 9), já no binário, se aplicam apenas dois valores (0 e 1). A formulação matemática no sistema binário é praticamente igual a da decimal, trocando apenas a multiplicação que antes era por 10 por 2.

$$N = \sum_{P=0}^{n-1} (A * 2^{P})$$

2FIGURA 2.18 - Fórmula do sistema binário FONTE: o autor.

Vejamos como exemplo o número em binário : N (101010)2

Podemos dizer, então, que o número binário é o somatório da multiplicação dos números elevados à potência conforme a quantidade de números binários, igual ao sistema decimal, a leitura ocorre da direita para esquerda, iniciando por 0, sendo efetuado o cálculo da mesma forma do sistema decimal.

$N = 1*25 + 0*24 + 1*2^{3} + 0*2^{2} + 1*2^{1} + 0*20$

$N = 1*25 = 1* 32 = 32$

0*24 = 0* 16 = 0

$1*2^3 = 1 * 8 = 8$

$0*2^2 = 0 * 4 = 0$

$1*2^1 = 1 * 2 = 2$

0*20 = 0 * 1 = 0

Total = 42

O número binário 101010 no sistema decimal corresponde a 4210.

Conversão de Decimal para Binário

O sistema decimal possui a combinação de 10 números conforme já apresentado, mas o CPU lê apenas números binários 0 e 1, dessa forma, é necessária a conversão do sistema decimal para o binário.

Para representar cada número, é necessário utilizar ao menos 4 dígitos binários, vejamos a conversão do número 7 na base 10 para binário.

$N= 0*2^3 + 1*2^2 + 1*2^1 + 1*20$

$N = 1*2^3 = 0 * 8 = 0$

$1*2^2 = 1 * 4 = 4$

$1*2^1 = 1 * 2 = 2$

1*20 = 1 * 1 = 1

Total = 710

Como o sistema decimal possui 10 números, podemos ter a variação de 10 conjuntos binários com 4 dígitos, conforme Tabela 2.2:

DECIMAL	BINÁRIO
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

2QUADRO 2.11 - undefined

Tabela de conversão decimal para binário

o autor.

Note que é possível termos 16 combinações utilizando os 4 dígitos binários, mas apenas 10 combinações são utilizadas para a representação dos números do sistema decimal, é importante ressaltar que a mesma combinação poderá ter um significado diferente, com base nas regras utilizadas na interpretação do sistema.

Sistema Hexadecimal

Por haver uma limitação no sistema decimal e por ser possível termos 16 combinações no sistema binário, foi desenvolvido o sistema hexadecimal.

> O sistema numérico de base 16, também chamado de hexadecimal ou hexa, usa 16 símbolos numéricos. Como existem apenas nove algarismos no nosso sistema alfanumérico (0 a 9), o sistema hexa usa letras em vez de números para valores maiores que nove .
>
> (NORTON, 1943 p. 107)

Um dígito hexadecimal corresponde a qualquer combinação possível do sistema binário, conforme mostra a Tabela 2.3. Também é considerado um intermediário entre o que podemos ver e o que o computador lê, no que se refere ao hardware.

DECIMAL	BINÁRIO	HEXADECIMAL
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

2QUADRO 3.11 - undefined

Representação numérica dos sistemas Decimal, Binário e Hexadecimal

o autor.

Qualquer conjunto binário pode ser representado no sistema hexadecimal, vamos pegar, por exemplo, o conjunto binário 01011101. Para efetuar a conversão, devemos dividir o conjunto binário em grupo de 4 algarismos, sendo feita da direita para a esquerda.

1101

Após a separação, consultamos a tabela 2.3, e verificamos qual dígito hexa corresponde ao conjunto binário.

0101 (binário) = 5 (hexa)

1101 (binário) = D (hexa)

Podemos nos deparar com um conjunto binário que, ao ser dividido, não forme um grupo de 4 algarismos, dessa forma, devemos preencher para a esquerdo com o número zero (0), para completar o grupo. Vamos utilizar como exemplo o conjunto binário 1111010111.

Novamente, vamos separar o conjunto binário em grupos de 4 algarismos, iniciando da direita para a esquerda.

1111

Observe, prezado(a) aluno(a), que o primeiro conjunto não é composto por 4 números, por isso, ele é completado com os números 0 a sua esquerda até completar os 4 algarismos, sendo composto da seguinte forma:

001111

Agora que temos todos os grupos completos, vamos consultar novamente a Tabela 2.3 e verificar qual dos dígitos hexa corresponde ao conjunto binário.

0011 (binário) = 3 (hexa)

1010 (binário) = A (hexa)

1111 (binário) = F (hexa)

O conjunto binário 1111010111$_2$, em hexadecimal, é 3AF$_{16}$

Alocação de Memória

Conforme visto na unidade anterior, a linguagem C armazena os seus valores na memória do computador durante a execução de um programa, essa alocação na memória acontece por meio do sistema hexadecimal, tanto números como letras são representados nesse sistema.

Para apresentar os caracteres para o usuário, é efetuada uma nova conversão, dessa vez do sistema hexadecimal para o ACSII (American Standard Code for Information Interchange), o qual é o padrão do Brasil, por suportar os caracteres especiais. No sistema ASCII, é possível termos até 255 combinações diferentes, vejamos a tabela 2.4 com alguns exemplos de caracteres.

CARACTER	DECIMAL	HEXADECIMAL	BINÁRIO
!	33	21	0010 0001

(	40	28	0010 1000
)	41	29	0010 1001
%	37	25	0010 0101
&	38	26	0010 0110
@	64	40	0100 0000

2QUADRO 4.11 - undefined

Exemplo de conjunto de caracteres ASCII

o autor.

Ao se manipularem os números, nos deparamos com situações que esses podem ser positivos ou negativos. Para diferenciar esse tipo de valor na alocação de memória, é inserido um número adicional no ínicio do conjunto binário, se o número for positivo, insere-se o 0, se for negativo, 1.

Por exemplo, o conjunto binário 101010 é equivalente a 42, se na execução do programa esse número for positivo, é inserido no início do conjunto o número 0, sendo assim, 00101010; mas se for negativo, será inserido o número 1, sendo também invertida a sequência da seguinte forma 11010101.

¶ Fique por dentro

Prezado(a) aluno(a), caso queira descobrir como são os demais caracteres em algum dos sistemas apresentados, decimal, binário ou hexadecimal, acesse o link e veja a tabela completa da ASCII: **equipe.nce.ufrj.br <http://equipe.nce.ufrj.br/adriano/algoritmos/apostila/tabascii.htm>** .

Listas Lineares

Uma lista linear é um vetor no qual os dados são agrupados com base no mesmo tipo de dado de maneira sequencial. Os elementos do vetor não precisam estar necessariamente na sequência física, alocados na memória, mas sim, logicamente, na ordem crescente no vetor. Podemos utilizar como exemplo o atendimento de um banco, várias pessoas estão espalhadas pelo local, mas a ordem de atendimento tem como base a senha sequencial de chegada.

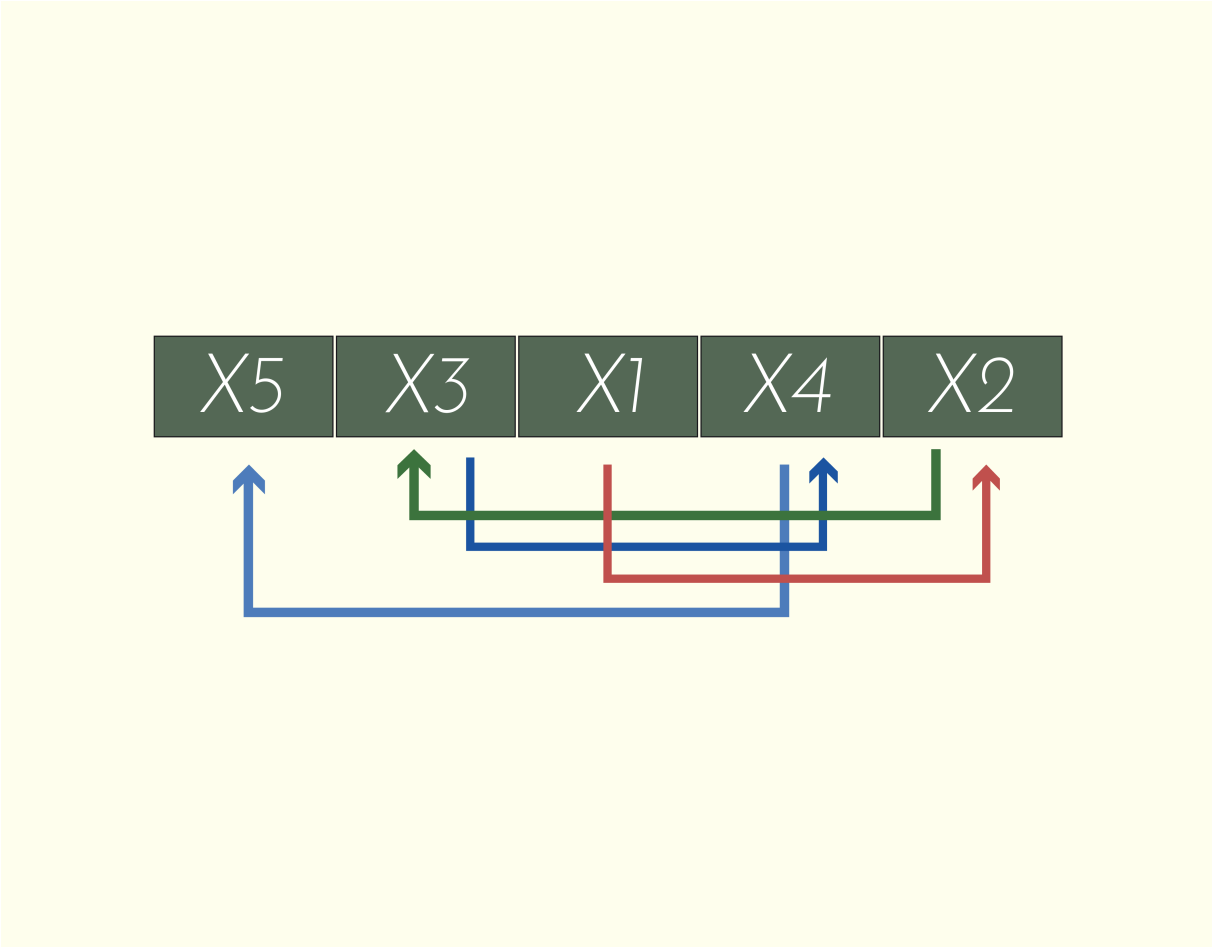
As listas lineares sempre dão sentido na ordem, por exemplo, do elemento na posição 3 do vetor para o elemento 4 e assim por diante. Temos vários tipos de listas, mas todas têm o mesmo princípio.

Listas Encadeadas

Conforme visto, uma lista é um vetor no qual são armazenados vários elementos, uma lista encadeada se refere a um vetor o qual é possível percorrer de forma ordenada, mesmo se os dados não estiverem em ordem dentro do vetor.

> Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista .
>
> (RANGEL, 2008)

Uma lista encadeada consiste em uma sequência de elementos fora de ordem, sendo que cada alocação do vetor é chamada de *nó da lista*. A lista se inicia com um elemento e um ponteiro que indica o próximo nó, dessa forma, a partir do primeiro elemento, podemos percorrer todo o vetor, seguindo o encadeamento. O último elemento aponta para *NULL*, o que sinaliza que não temos mais elementos.



2FIGURA 3.18 - Exemplo de como percorrer uma lista encadeada FONTE: o autor.

No exemplo apresentado na Figura 2.3, ao se percorrer o vetor, a leitura se inicia pela alocação X1, junto ao valor alocação, é armazenado um ponteiro que informa qual será a próxima posição dentro do vetor. No caso apresentado, a leitura será feita da seguinte forma:

X1 → X2 → X3 → X4 → X5

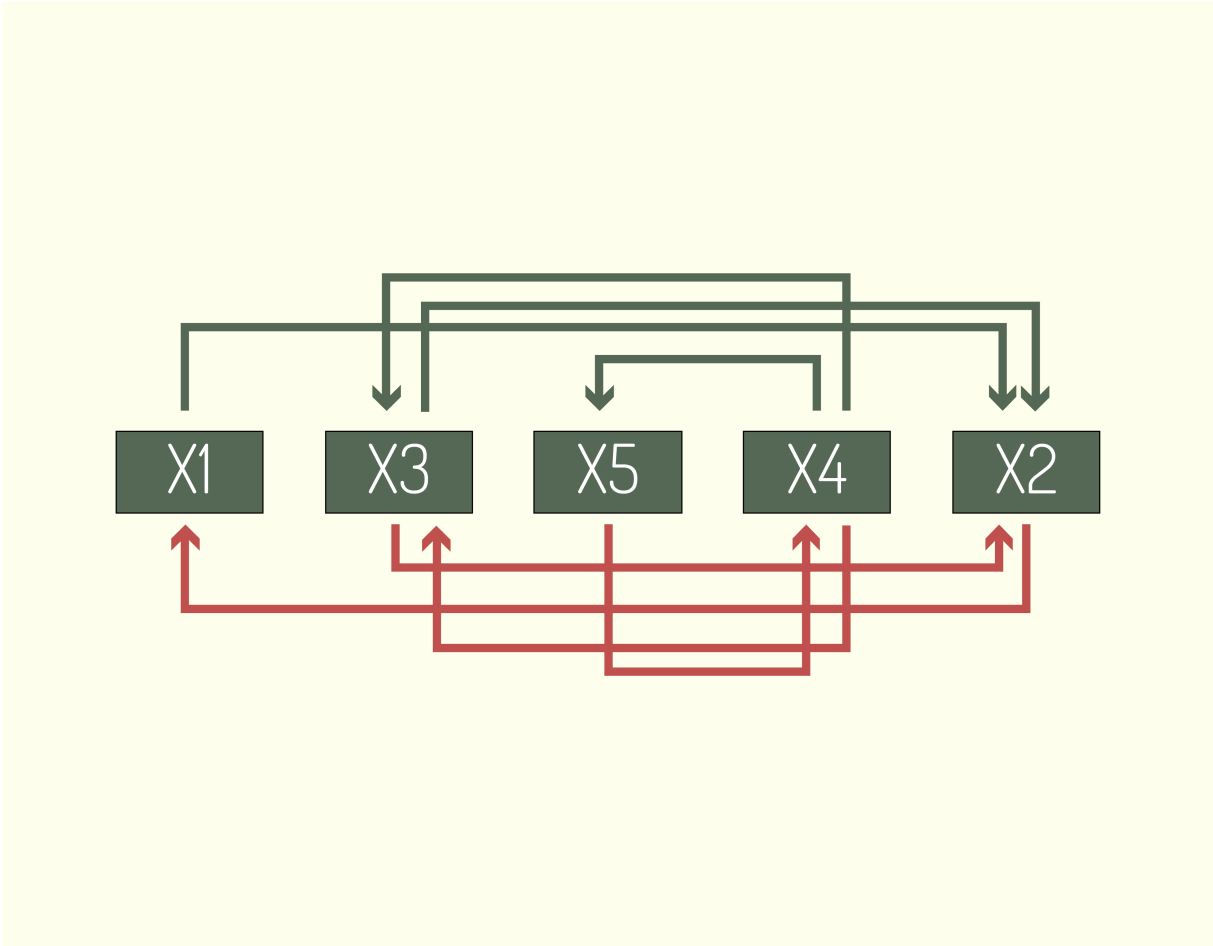
Listas Duplamente Encadeadas

A estrutura apresentada anteriormente é conhecida como lista encadeada simples, na qual se armazena apenas o ponteiro do próximo elemento a ser lido. Um problema que encontramos nesse tipo de estrutura é que sempre somos levado para frente, nunca para trás, caso precise acessar o elemento anterior, isso não é possível, da mesma forma que somos obrigados a percorrer toda a lista para encontrar o elemento desejado.

Caso seja necessário ler um elemento anterior, temos que concluir toda a leitura e reiniciá-la. Rangel (2008 p. 18) afirma que uma solução é a lista duplamente encadeada:

Nelas, cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior. Desta forma, dado um elemento, podemos acessar ambos os elementos adjacentes: o próximo e o anterior. Se tivermos um ponteiro para o último elemento da lista, podemos percorrer a lista em ordem inversa, bastando acessar continuamente o elemento anterior, até alcançar o primeiro elemento da lista, que não tem elemento anterior (o ponteiro do elemento anterior vale NULL).

Com a lista duplamente encadeada, é possível percorrer toda a estrutura, chegando ao último elemento, podemos também encerrar ou efetuar a leitura inversa dos elementos e, em qualquer momento, podemos voltar para o elemento anterior.



2FIGURA 4.18 - lista duplamente encadeada FONTE: o autor.

Na Figura 2.4, vemos como é percorrida a lista ao ser iniciada pelo elemento que está armazenado no X1, ela pode então ser lida da seguinte forma:

ELEMENTO ANTERIOR	ELEMENTO ATUAL	PRÓXIMO ELEMENTO

NULL	X1	X2
X1	X2	X3
X2	X3	X4
X3	X4	X5
X4	X5	NULL

2QUADRO 5.11 - undefined

Composição dos elementos na lista duplamente encadeada

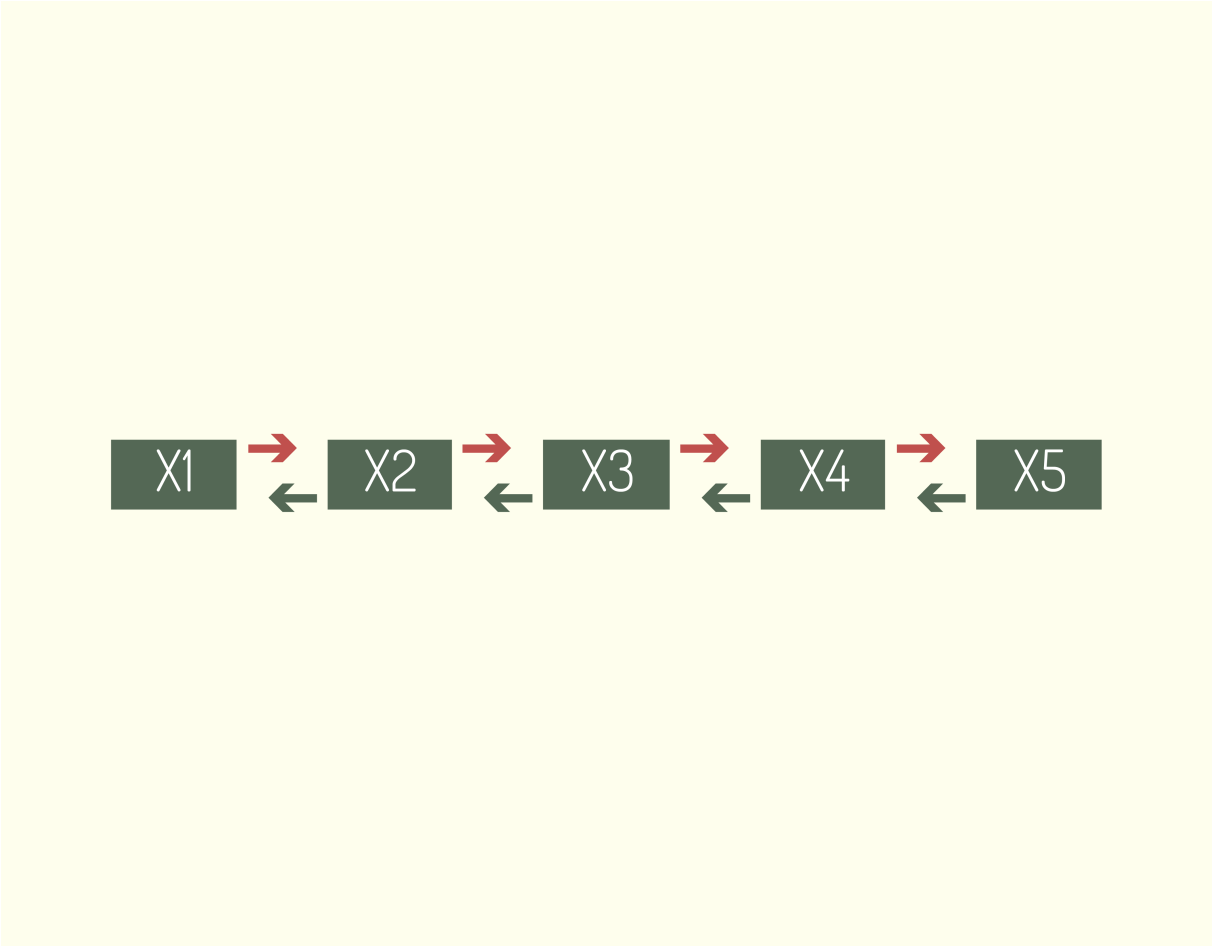
o autor.

Perceba, caro(a) aluno(a), que se estivermos no elemento X3, podemos retornar para o elemento X2, mas se quisermos ir do X4 para o X1, temos que percorrer todos os demais elementos até chegar no nosso destino, sendo também possível ir para frente ou para trás a qualquer momento.

Listas Ordenadas

Compreendemos que ao se utilizar a lista duplamente encadeada temos uma maior flexibilidade ao lidar com os dados, mas ainda nos deparamos com uma situação, os dados estão fora de ordem, isso gera um consumo mais elevado dos recursos computacionais. Tanto na lista encadeada simples quanto na dupla, encontramos a mesma situação, uma solução para isso é a ordenação dos elementos.

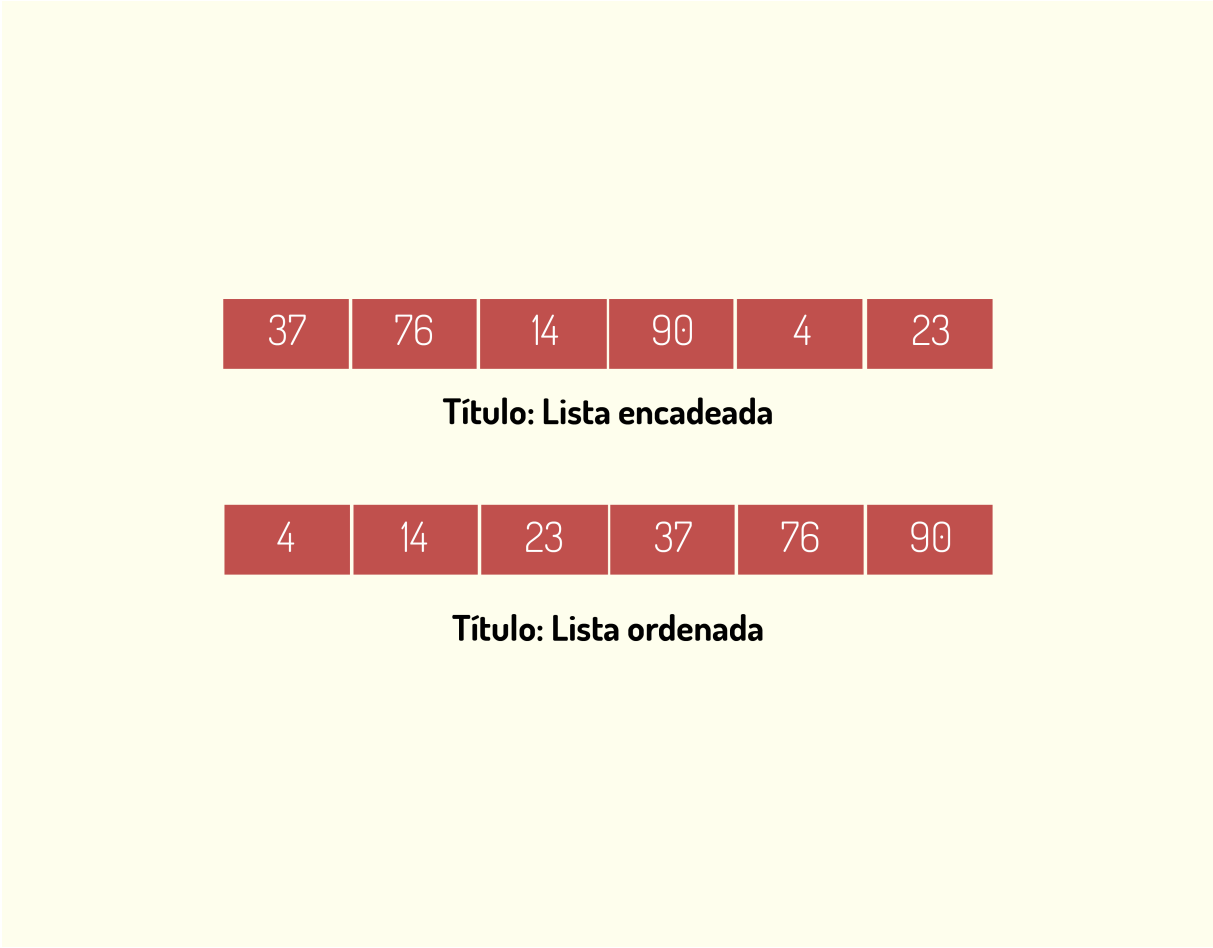
A ordenação da lista faz sentido se pensarmos que temos que percorrer toda ou parte dela para encontrar um elemento. Podemos ordenar a lista de modo crescente ou decrescente, alfabética de A - Z ou Z - A, dependendo do tipo de aplicação, assim, gerando agilidade no retorno dos resultados.



2FIGURA 5.18 - Lista duplamente ordenada FONTE: o autor.

Com a lista ordenada, o sistema pode encontrar mais fácil os elementos utilizando menos recursos, perceba, caro(a) aluno(a), que o caminho a ser percorrido para encontrar o próximo elemento é muito menor do que quando tínhamos apenas a lista encadeada. Vejamos alguns exemplos de ordenação de listas, utilizando elementos numéricos e caracteres.

Exemplo de lista com elementos numéricos:



2FIGURA 6.18 - Lista de elementos numéricos FONTE: o autor

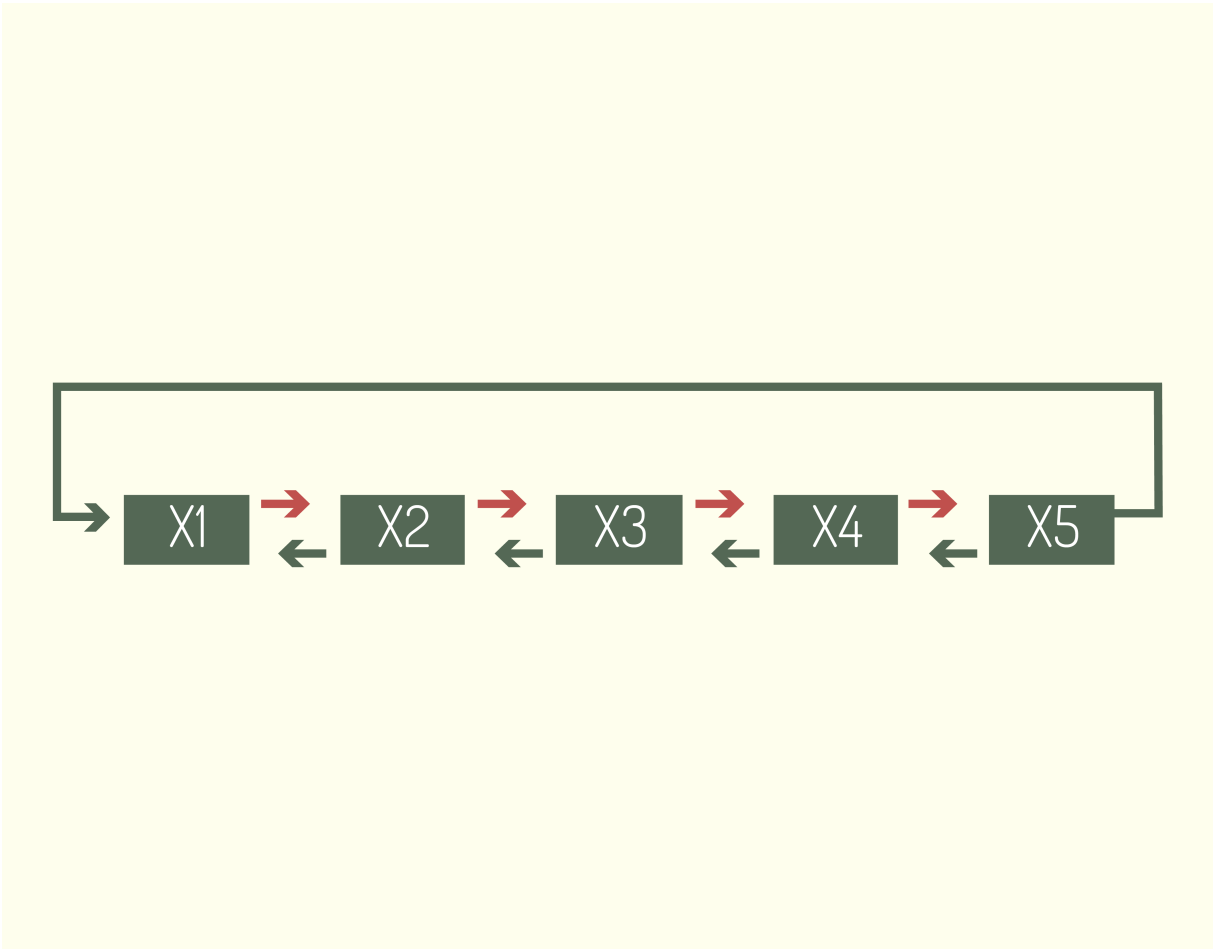
Exemplo de lista com caracteres:

2FIGURA 7.18 - Lista com caracteres FONTE: o autor.

Listas Circulares

As listas circulares foram pensadas para que a leitura chegue ao fim e possa retornar ao início sem que seja necessário realizar diversas iterações ou reiniciar a busca. O método de list circular pode ser aplicado em qualquer conceito estudado até o momento, listas simples, duplas e ordenadas.

Ao aplicar a busca em uma lista simples ou dupla ordenada, as interações partem do ponteiro na posição inicial e vão até o último, antes a leitura se encerrava ou efetuava a busca inversa, no caso da lista duplamente ordenada, aplicando o conceito de listas circulares, o ponteiro do último elemento informa que o próximo elemento é o início da lista.

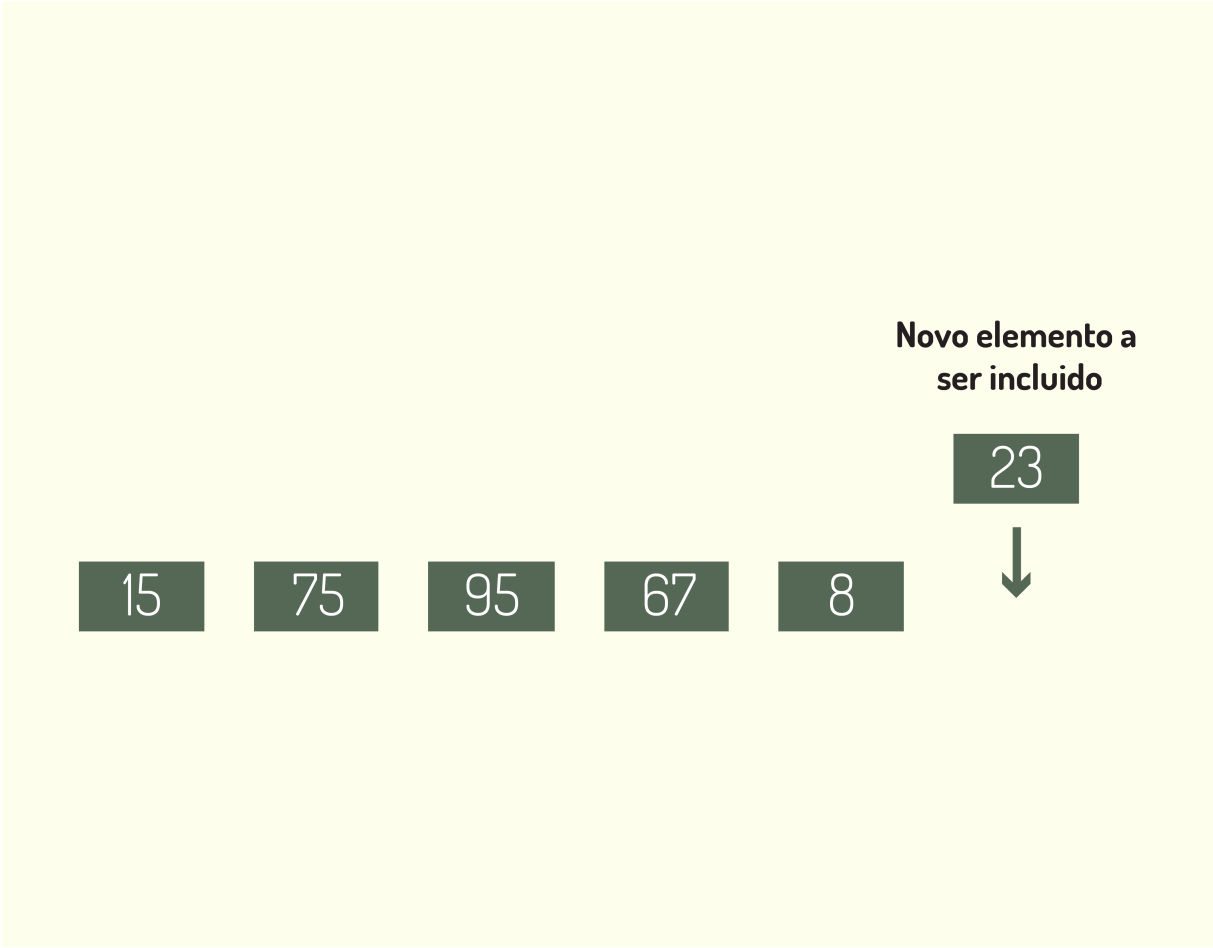


2FIGURA 8.18 - Lista duplamente ordenada circular FONTE: o autor.

Dessa forma, é possível visitar todos os elementos a partir do ponteiro inicial até alcançá-lo novamente. A lista circular só retorna para o início através do último elemento, para isso, é preciso percorrê-la por completo. O elemento inicial possui o ponteiro que irá identificar o próximo elemento, mas o que identifica o elemento anterior irá identificar como NULL.

Inclusão de dados em listas

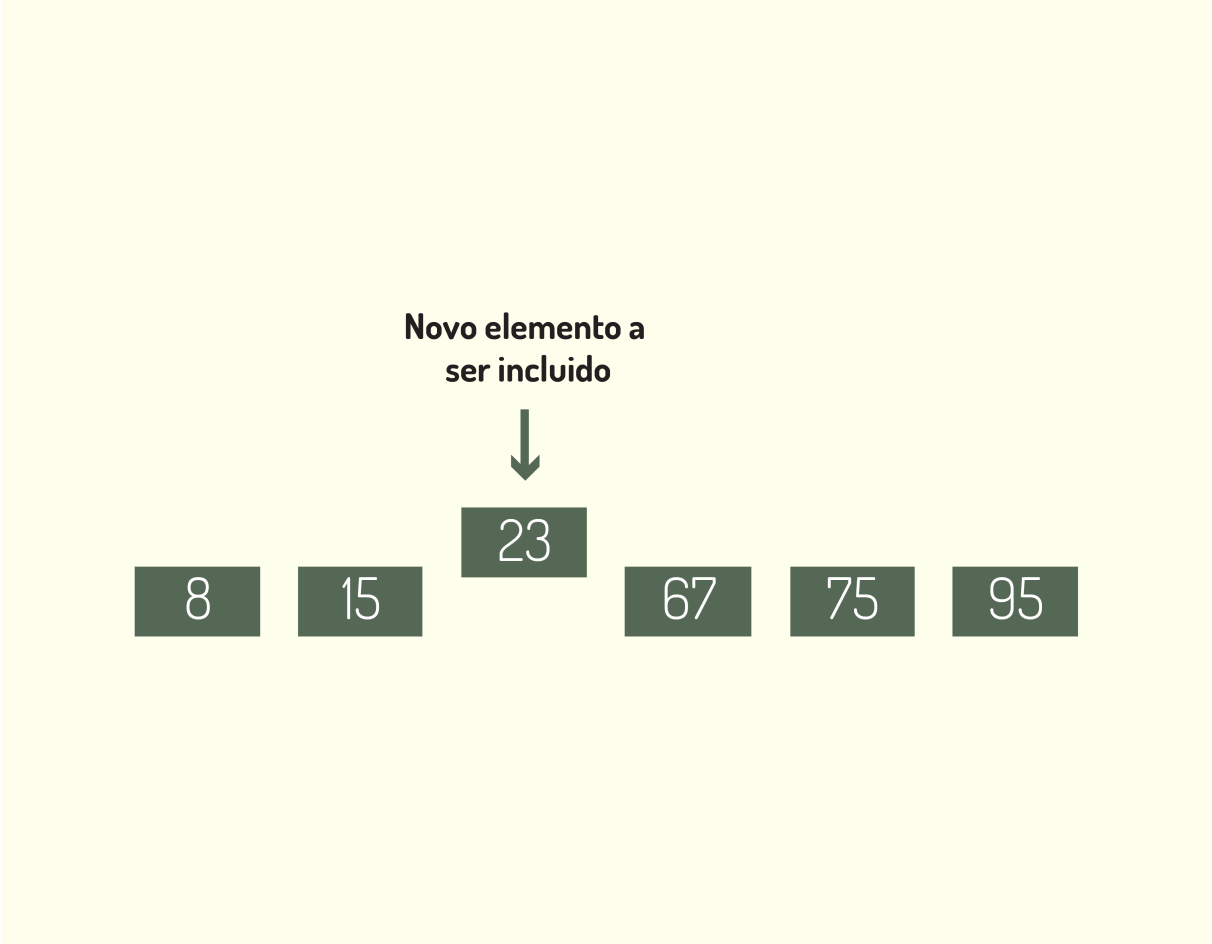
A inserção de um novo elemento em uma lista, sendo ela simples ou dupla, terá uma variação de posição caso esteja ordenada ou apenas encadeada. Em uma lista sem ordenação, o elemento sempre será inserido no final do vetor; em uma lista ordenada, antes de efetuar a inclusão do elemento, é efetuada uma leitura para identificar a posição correta, a lista pode estar ordenada de forma crescente ou decrescente.



2FIGURA 9.18 - Inclusão de elemento em uma lista não ordenada FONTE: o autor.

A inclusão do número 23 será feita no final da lista, pois esta não está ordenada.

Na Figura 2.10, a lista é duplamente encadeada, veja onde o elemento é inserido:



2FIGURA 10.18 - Inclusão de elemento em uma lista duplamente ordenada FONTE: o autor.

Com a lista ordenada, antes da inclusão, é efetuada uma leitura para encontrar o local onde será inserido o elemento na ordem correta. Podemos ver nas Tabelas 2.5 e 2.6 que todos os elementos a partir do número 67 ocuparam uma nova posição no vetor:

Posição no vetor	0	1	2	3	4
Elemento	8	15	67	75	95

2QUADRO 6.11 - undefined

Vetor antes da inclusão

o autor.

Posição no vetor	0	1	2	3	4	5
Elemento	8	15	23	67	75	95

2QUADRO 7.11 - undefined

Vetor antes da inclusão

o autor.

Remoção de dados em listas

A remoção de qualquer elemento da lista acontece da mesma forma da inclusão, a diferença é que a ordenação não será um diferencial, pois, se o elemento é removido do final da lista, esta não sofre nenhum tipo de movimentação no vetor; caso o elemento esteja no início ou no meio do vetor, haverá movimentos na lista.

Vamos utilizar como exemplo a lista apresentada anteriormente:

Posição no vetor	0	1	2	3	4	5
Elemento	8	15	23	67	75	95

2QUADRO 8.11 - undefined

Vetor antes da remoção

o autor.

Posição no vetor	0	1	2	3	4
Elemento	15	23	67	75	95

2QUADRO 9.11 - undefined

Vetor após a remoção

o autor.

Veja, caro(a) aluno(a), que, ao excluir o número 8, todos os demais assumiram posições novas, caso o elemento removido tivesse sido o número 23, todos os elementos cuja posição fosse maior que a do número 23 teriam sofrido modificações, já os elementos anteriores ao número 23 não.

Implementação de listas em linguagem C

Após compreendermos como funcionam as listas e suas particularidades, além de vermos também como pode-se inserir e remover conteúdo de uma lista, vejamos a seguir um exemplo de código fonte em linguagem C:

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

struct Node{
        int num;
        struct Node *prox;
};
typedef struct Node node;

int tam;

void inicia(node *LISTA);
int menu(void);
void opcao(node *LISTA, int op);
node *criaNo();
void insereFim(node *LISTA);
void insereInicio(node *LISTA);
void exibe(node *LISTA);
void libera(node *LISTA);
void insere (node *LISTA);
node *retiraInicio(node *LISTA);
node *retiraFim(node *LISTA);
node *retira(node *LISTA);


int main(void)
{
        node *LISTA = (node *) malloc(sizeof(node));
        if(!LISTA){
                printf(“Sem memoria disponivel!\n”);
                exit(1);
        }else{
        inicia(LISTA);
        int opt;

        do{
                opt=menu();
                opcao(LISTA,opt);
        }while(opt);

        free(LISTA);
        return 0;
        }
}

void inicia(node *LISTA)
{
        LISTA->prox = NULL;
        tam=0;
}

int menu(void)
{
        int opt;

        printf(“Escolha a opcao\n”);
        printf(“0. Sair\n”);
        printf(“1. Zerar lista\n”);
        printf(“2. Exibir lista\n”);
        printf(“3. Adicionar node no inicio\n”);
        printf(“4. Adicionar node no final\n”);
        printf(“5. Escolher onde inserir\n”);
        printf(“6. Retirar do inicio\n”);
        printf(“7. Retirar do fim\n”);
        printf(“8. Escolher de onde tirar\n”);
        printf(“Opcao: “); scanf(“%d”, &opt);

        return opt;
}

void opcao(node *LISTA, int op)
{
        node *tmp;
        switch(op){
                case 0:
                        libera(LISTA);
                        break;
```

```
                case 1:
                        libera(LISTA);
                        inicia(LISTA);
                        break;

                case 2:
                        exibe(LISTA);
                        break;

                case 3:
                        insereInicio(LISTA);
                        break;

                case 4:
                        insereFim(LISTA);
                        break;

                case 5:
                        insere(LISTA);
                        break;

                case 6:
                        tmp= retiraInicio(LISTA);
                        printf("Retirado: %3d\n\n", tmp->num);
                        break;

                case 7:
                        tmp= retiraFim(LISTA);
                        printf("Retirado: %3d\n\n", tmp->num);
                        break;

                case 8:
                        tmp= retira(LISTA);
                        printf("Retirado: %3d\n\n", tmp->num);
                        break;

                default:
                        printf("Comando invalido\n\n");
        }
}

int vazia(node *LISTA)
{
        if(LISTA->prox == NULL)
                return 1;
        else
                return 0;
}

node *aloca()
{
        node *novo=(node *) malloc(sizeof(node));
        if(!novo){
                printf("Sem memoria disponivel!\n");
                exit(1);
        }else{
                printf("Novo elemento: "); scanf("%d", &novo->num);
                return novo;
        }
}

void insereFim(node *LISTA)
{
        node *novo=aloca();
        novo->prox = NULL;

        if(vazia(LISTA))
                LISTA->prox=novo;
        else{
                node *tmp = LISTA->prox;

                while(tmp->prox != NULL)
                        tmp = tmp->prox;

                tmp->prox = novo;
        }
```

```
        tam++;
}

void insereInicio(node *LISTA)
{
        node *novo=aloca();
        node *oldHead = LISTA->prox;

        LISTA->prox = novo;
        novo->prox = oldHead;

        tam++;
}

void exibe(node *LISTA)
{
        system(“cls”);
        if(vazia(LISTA)){
                printf(“Lista vazia!\n\n”);
                return ;
        }

        node *tmp;
        tmp = LISTA->prox;
        printf(“Lista:”);
        while( tmp != NULL){
                printf(“%5d”, tmp->num);
                tmp = tmp->prox;
        }
        printf(“\n      “);
        int count;
        for(count=0 ; count < tam ; count++)
                printf(“  ^  “);
        printf(“\nOrdem:”);
        for(count=0 ; count < tam ; count++)
                printf(“%5d”, count+1);

        printf(“\n\n”);
}

void libera(node *LISTA)
{
        if(!vazia(LISTA)){
                node *proxNode,
                          *atual;

                atual = LISTA->prox;
                while(atual != NULL){
                        proxNode = atual->prox;
                        free(atual);
                        atual = proxNode;
                }
        }
}

void insere(node *LISTA)
{
        int pos,
                count;
        printf(“Em que posicao, [de 1 ate %d] voce deseja inserir: “, tam);
        scanf(“%d”, &pos);

        if(pos>0 && pos <= tam){
                if(pos==1)
                        insereInicio(LISTA);
                else{
                        node *atual = LISTA->prox,
                                 *anterior=LISTA;
                        node *novo=aloca();

                        for(count=1 ; count < pos ; count++){
                                        anterior=atual;
                                        atual=atual->prox;
                        }
                        anterior->prox=novo;
                        novo->prox = atual;
                        tam++;
```

```
                }

        }else
                printf(“Elemento invalido\n\n”);
}

node *retiraInicio(node *LISTA)
{
        if(LISTA->prox == NULL){
                printf(“Lista ja esta vazia\n”);
                return NULL;
        }else{
                node *tmp = LISTA->prox;
                LISTA->prox = tmp->prox;
                tam--;
                return tmp;
        }

}

node *retiraFim(node *LISTA)
{
        if(LISTA->prox == NULL){
               printf(“Lista ja vazia\n\n”);
               return NULL;
        }else{
                node *ultimo = LISTA->prox,
                      *penultimo = LISTA;

                while(ultimo->prox != NULL){
                        penultimo = ultimo;
                        ultimo = ultimo->prox;
               }

                penultimo->prox = NULL;
                tam--;
                return ultimo;
        }
}

node *retira(node *LISTA)
{
        int opt,
               count;
        printf(“Que posicao, [de 1 ate %d] voce deseja retirar: “, tam);
        scanf(“%d”, &opt);

        if(opt>0 && opt <= tam){
               if(opt==1)
                        return retiraInicio(LISTA);
                else{
                        node *atual = LISTA->prox,
                                 *anterior=LISTA;

                       for(count=1 ; count < opt ; count++){
                               anterior=atual;
                               atual=atual->prox;
                       }

                anterior->prox=atual->prox;
                tam--;
                return atual;
               }

        }else{
               printf(“Elemento invalido\n\n”);
               return NULL;
        }
}
```

COMO fazer uma lista em C -Implementação completa (inserindo e retirando nós de qualquer posição). C progressivo. **www.cprogressivo.net <http://www.cprogressivo.net/2013/10/Como-fazer-uma-lista-em-C.html>** .

-

Árvores

O conceito do que é uma árvore no mundo real é bem simples, nesse sentido, a árvore do ambiente computacional tem algumas semelhanças com a do mundo real. A estrutura de dados em árvore é utilizada por diversos sistemas e programas para organização das informações armazenadas na memória secundária e principal no momento de execução.

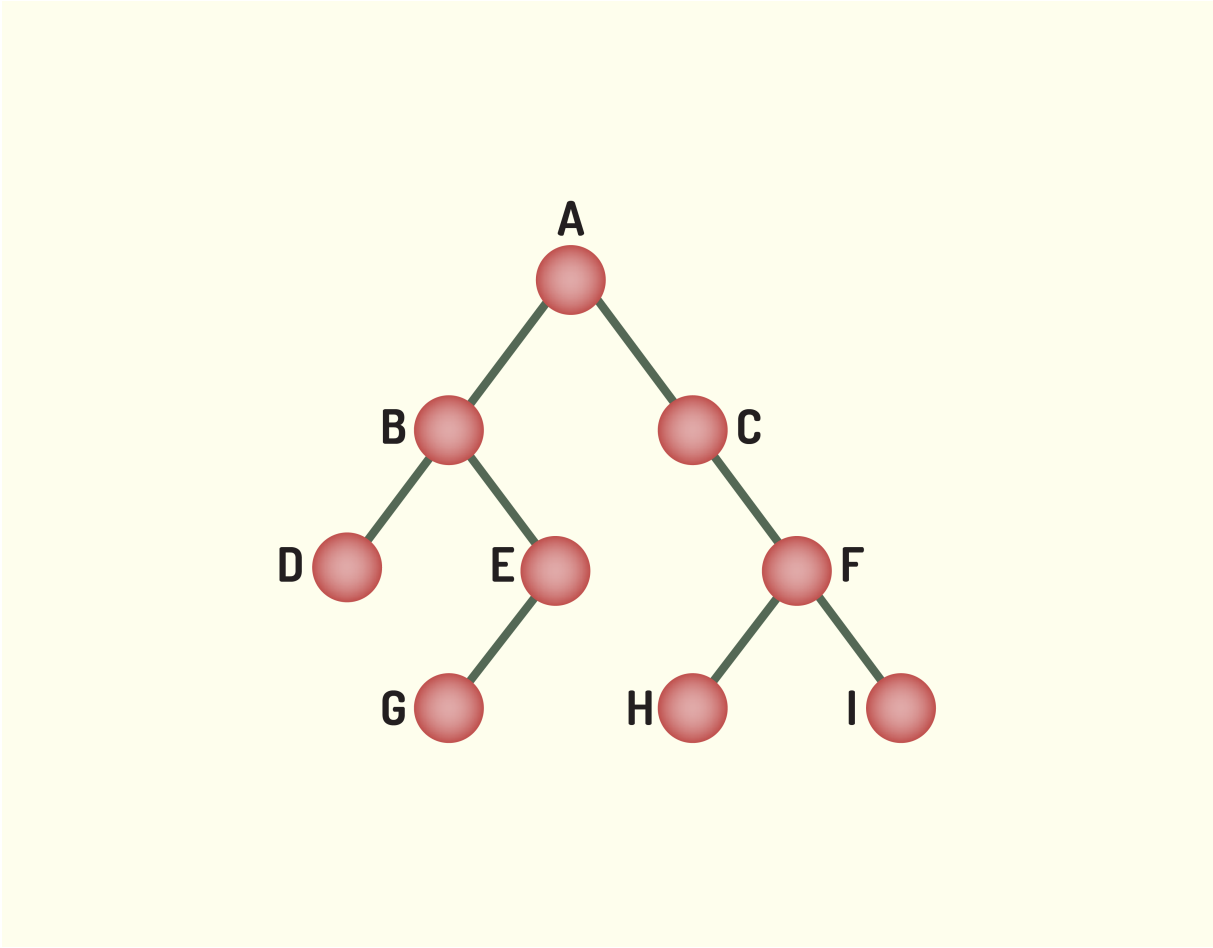
A vantagem de se utilizar essa estrutura está no fato de ela permitir acesso fácil e rápido aos dados nas árvores, as quais são aplicadas de forma binária, conforme veremos.

Árvores Binária

Todos nós já vimos e sabemos como é uma árvore, mas no ambiente computacional ela é representada de um modo um pouco diferente, já que utilizamos a sua estrutura de modo binário. Segundo Tenenbaum (1995, p. 303),

> Uma árvore binária é um conjunto finito de elementos que está vazio ou é particionado em três subconjuntos. O primeiro subconjunto contém um único elemento, chamado raiz da árvore. Os outros dois subconjuntos são em si mesmos árvores binárias, chamadas subárvores esquerda e direita da árvore original. Uma subárvore esquerda ou direita pode estar vazia. Cada elemento de uma árvore binária é chamado nó da árvore.

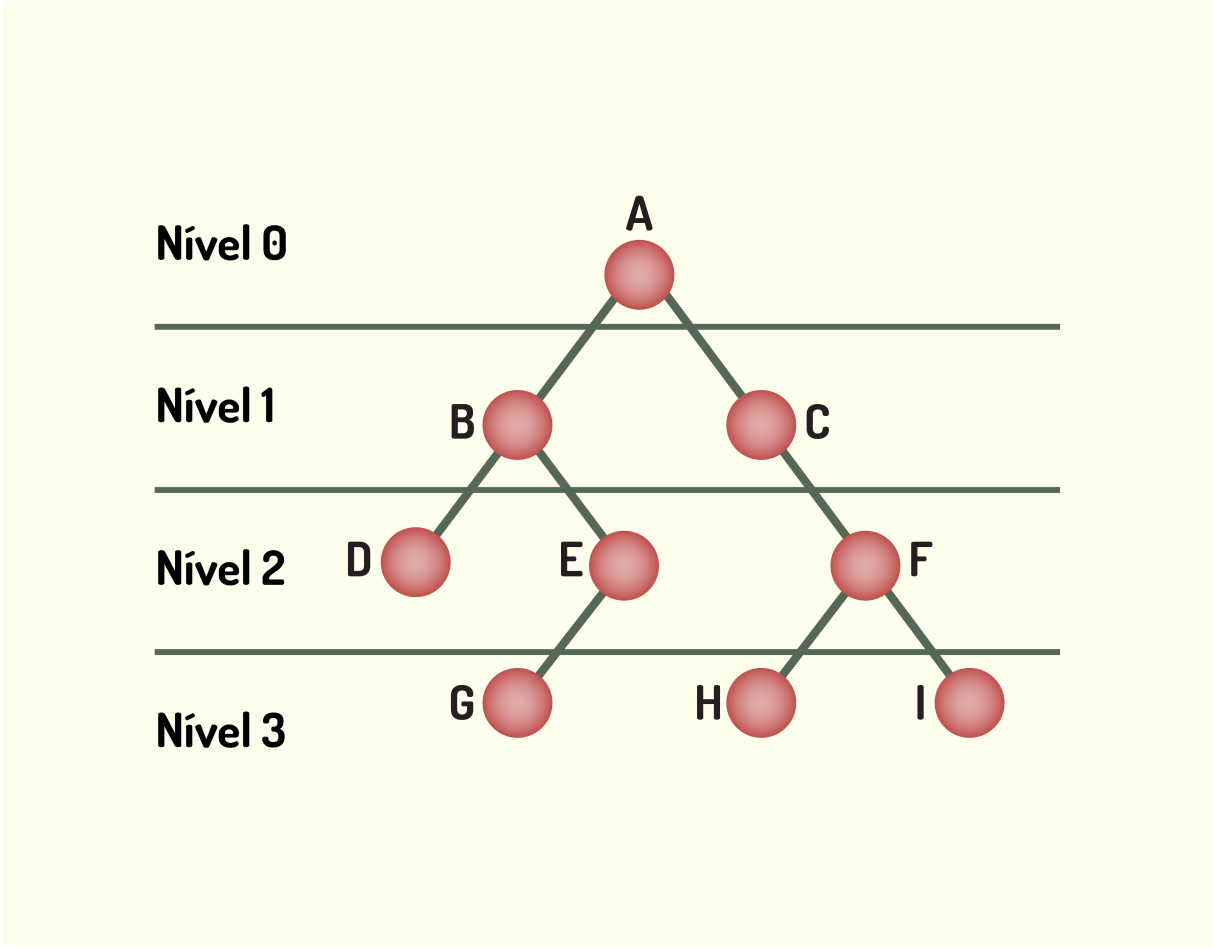
Uma árvore binária é caracterizada pela composição de uma raiz e 2 subárvores (direita e esquerda), podemos também chamar a raiz de pai, e cada subárvore, de filho à esquerda e filho à direita. Uma árvore binária tem no máximo 2 filhos a partir do pai.



2FIGURA 11.18 - Exemplo de árvore binária FONTE: o autor.

Na Figura 2.11, podemos observar que, ao contrário de uma árvore convencional, em que a raiz se inicia na parte inferior e tende a ir para cima, em uma árvore binária computacional, a raiz se inicia pela parte superior e se subdivide para baixo. As letras A, B, C, D, E, F, G, H e I são consideradas nós ou pais da árvore: um pai pode conter filhos ou não, o nó que não possuir um filho é considerado vazio, sendo chamado de folha da árvore, nesse caso, os elementos D, G, H e I são folhas da árvore apresentada.

Para identificar em qual nível (altura) da árvore se encontra, é contada a quantidade de vezes que se desce, os níveis se iniciam por 0 e é somado + 1 para cada nível que se desce com base no nível anterior, não temos um limite de níveis:



2FIGURA 12.18 - Identificação do nível da árvore FONTE: o autor.

Podemos agora identificar qual o nível da árvore, seus nós e folhas e também quais são os pais e seus filhos:

Nós no nível 0	A
Nós no nível 1	B, C
Nós no nível 2	D, E, F
Nós no nível 3	G, H, I
Nós	A, B, C, E, F
Folhas	D, G, H, I

2QUADRO 10.11 - undefined

Identificação dos níveis, nós e folhas

o autor.

ELEMENTO	PAI	FILHO ESQUEDA	FILHO DIREITA	TIPO
A	-	B	C	Raiz
B	A	D	E	Nó
C	A	-	F	Nó
D	B	-	-	Folha

E	B	G	-	Nó
F	C	H	I	Nó
G	E	-	-	Folha
H	F	-	-	Folha
I	F	-	-	Folha

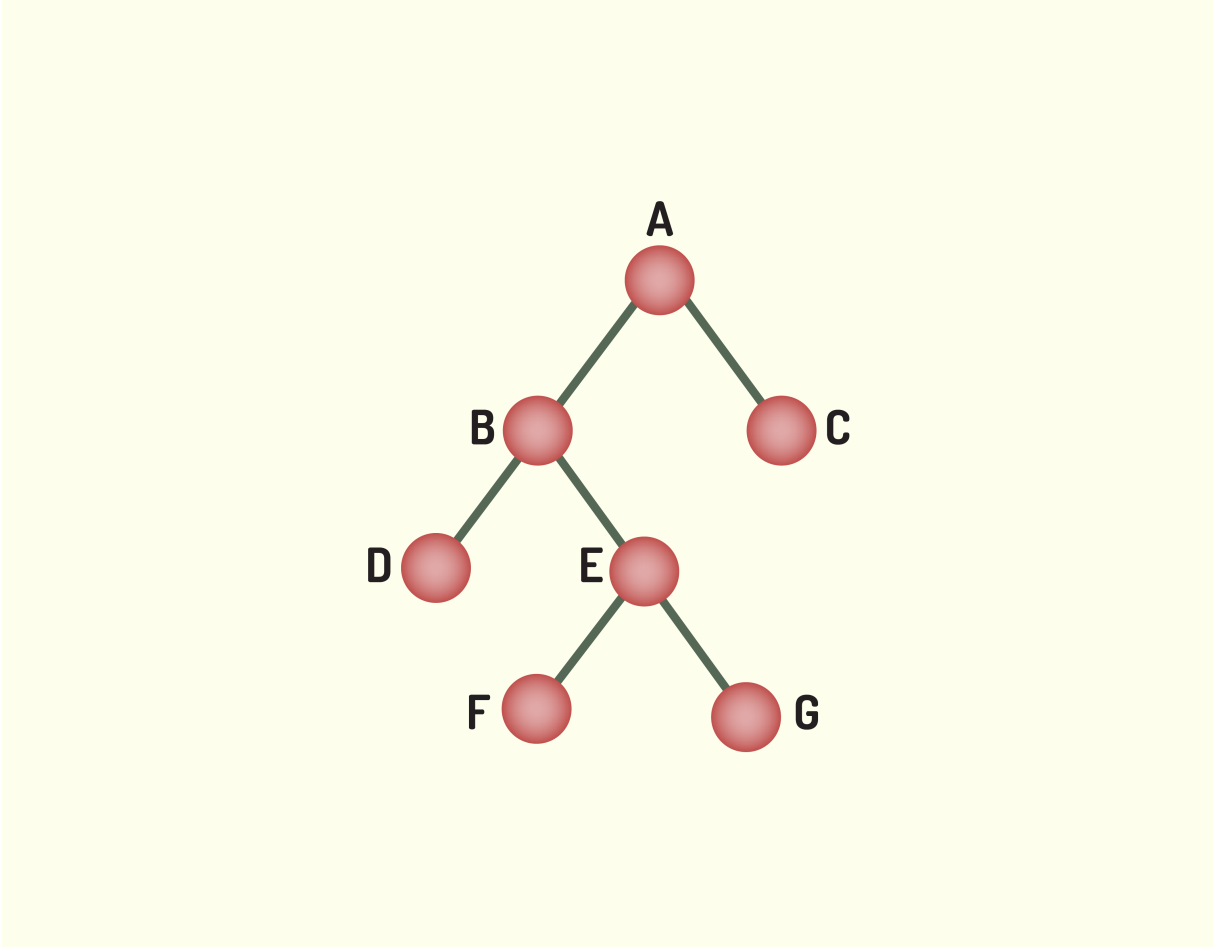
2QUADRO 11.11 - undefined

Identificação dos elementos na árvore binária

o autor.

Árvores Estritamente Binárias

Uma árvore estritamente binária tem como base os mesmos princípios da árvore binária comum, sendo composta por uma raiz, nós e folhas. Para que uma árvore seja totalmente binária, deve-se considerar a seguinte afirmação: é considerada uma árvore estritamente binária se todos os nós que não são folhas possuírem sempre duas subárvores, sendo a esquerda e a direita não vazias. Vejamos um exemplo.



2FIGURA 13.18 - Exemplo de árvore estritamente binária FONTE: o autor.

Compreenda, prezado(a) aluno(a), que todos os nós, exceto as folhas, têm um filho à esquerda e outro à direita, se tivermos uma árvore estritamente binária, conseguimos encontrar a quantidade de nós utilizando a quantidade de folhas com a seguinte fórmula n = (2 * f) - 1, sendo que F é o número de folhas da árvore. Vejamos a resolução da fórmula:

Sabemos que a nossa árvore possui 4 nós que são folhas C, D, F e G:

N = (2 * 4) - 1

Após realizar a multiplicação:

N = 8 - 1

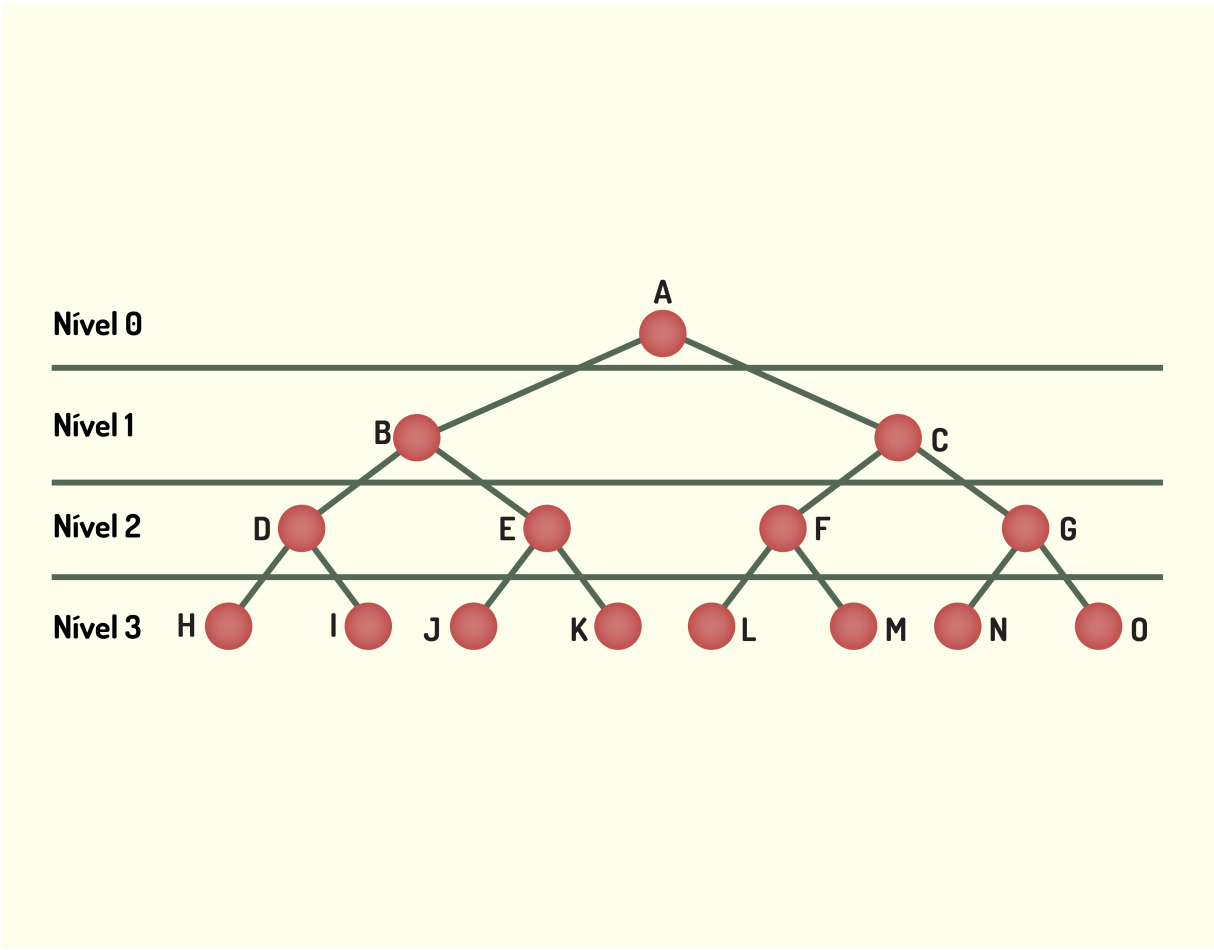
Ao efetuarmos a subtração teremos o seguinte resultado:

N = 7

Se analisarmos a nossa árvore binária, veremos que a quantidade de nós é exatamente 7: A, B, C, D, E, F e G.

Árvores Binária Completa

Uma árvore binária completa nada mais é que uma árvore estritamente binária, em que, em seu último nível, todos os elementos são folhas e todos os nós são dois filhos.



2FIGURA 14.18 - Exemplo de árvore binária completa FONTE: o autor.

Observe, caro(a) aluno(a), que, no último nível, temos os elementos H, I J, K, L, M, N e O, todos são folhas, pois não têm nenhum filho à esquerda ou à direita, já os demais elementos possuem dois filhos cada.

Implementação de árvore binária em linguagem C

Após compreendermos como funcionam as árvores binárias e suas particularidades, vejamos a seguir um exemplo de código fonte em linguagem C:

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

                int arvore[21];
                int i,j,aux;
                int resp=0,troca,busca;

int menu (){
                 system("cls");
                           printf("\n_____________________________________________________________\n\n");
                printf("\t\t|------------>MENU<-----------|");
                printf("\n\t\t|_____________________________|\n");
                printf("\t\t| 1 - Inserir numeros       |\n");
                printf("\t\t| 2 - Exibir numeros inseridos |\n");
                printf("\t\t| 3 - Exibir arvore binaria |\n");
                printf("\t\t| 4 - Sair                  |");
                printf("\n\t\t|_____________________________|\n");
                printf("\t\t\n\t\t\t Digite sua opcao: \a");
                scanf("%d",&resp);
                return resp;

}

void  comandos(int resp){
        if(resp==0){ resp = menu();}

                switch(resp){
                //insere numeros
                case 1:
                system("cls");
                printf("\n_____________________________________________________\n\n");
                printf(" -->>Informe os números a serem inseridos na arvore\n\n");
                for (i=0; i<15; i++){
                printf("\t\tPosicao     %d : ",i+1);
                scanf("%d",&arvore[i]);
                }
                printf("\n_____________________________________________________\n\n");
                                        comandos(0);
                break;

                case 2:
                system("cls");
                printf("\n_____________________________________________________\n\n");
                printf(" -->>Elementos no vetor da arvore!\n\n");
                for (i=0; i<15; i++){
                printf("\n\t -->> POSICAO %d  =  %d",i,arvore[i]);
                }
                printf("\n_____________________________________________________\n\n");
                system("pause");
                comandos(0);
                break;

                case 3:
                system("cls");
                printf("\n_____________________________________________________\n\n");
                                                printf("\n\n -->>ARVORE BINARIA!\n\n");
                printf("                       %d\n",arvore[0]);
                printf("               / \\ \n");
                printf("               /   \\ \n");
                printf("               /       \\ \n");
                printf("               /       \\ \n");
                printf("               /       \\ \n");
                printf("               /               \\ \n");
                printf("               /               \\ \n");
                printf("               /               \\ \n");
                printf("        %d                     %d \n",arvore[1],arvore[2]);
                printf("        /  \\                   /  \\  \n");
                printf("        /       \\              /       \\  \n");
                printf("        /       \\              /       \\  \n");
                printf("        %d      %d      %d      %d \n",arvore[3],arvore[4],arvore[5],arvore[6]);
                printf("        / \\    / \\    / \\    / \\  \n");
                printf("        /   \\  /   \\  /   \\  /   \\  \n");
                printf("        %d      %d  %d  %d   %d   %d  %d   %d \n",arvore[7], arvore[8],arvore[9],arvore[10], arvore[11
                printf("\n_____________________________________________________\n\n");
                system("pause");
                                                comandos(0);
```

```
                break;

                //finaliza
                case 4:
                resp = 0;
                system("cls");
                printf("\n_____\n");
                printf(" -->> Ate Logo!");
                printf("\n_____\n\n\n");
                }

}

main(){
                for (i = 0; i<21; i++){
                arvore[i] = i;
                }
                comandos(0);   }
```

Pilhas e Filas

As estruturas de dados pilha e fila são consideradas com lista lineares, mas com características próprias. As funções básicas como incluir elementos, remover elementos ou buscar estão presentes nessa estrutura. O que irá diferenciar essas listas das que já vimos até o momento é o seu modo de manutenção.

Essas estruturas são muito utilizadas na busca de dados em uma árvore ou em um grafo, em que se utiliza o conceito de busca em largura e em profundidade.

¶ Fique por dentro

A busca em profundidade se utiliza de uma pilha e se inicia pelo nó raiz e desce pela árvore até o último nível ou até encontrar uma folha, após isso, volta na estrutura da árvore e vai visitando os demais nós até efetuar a busca completa.

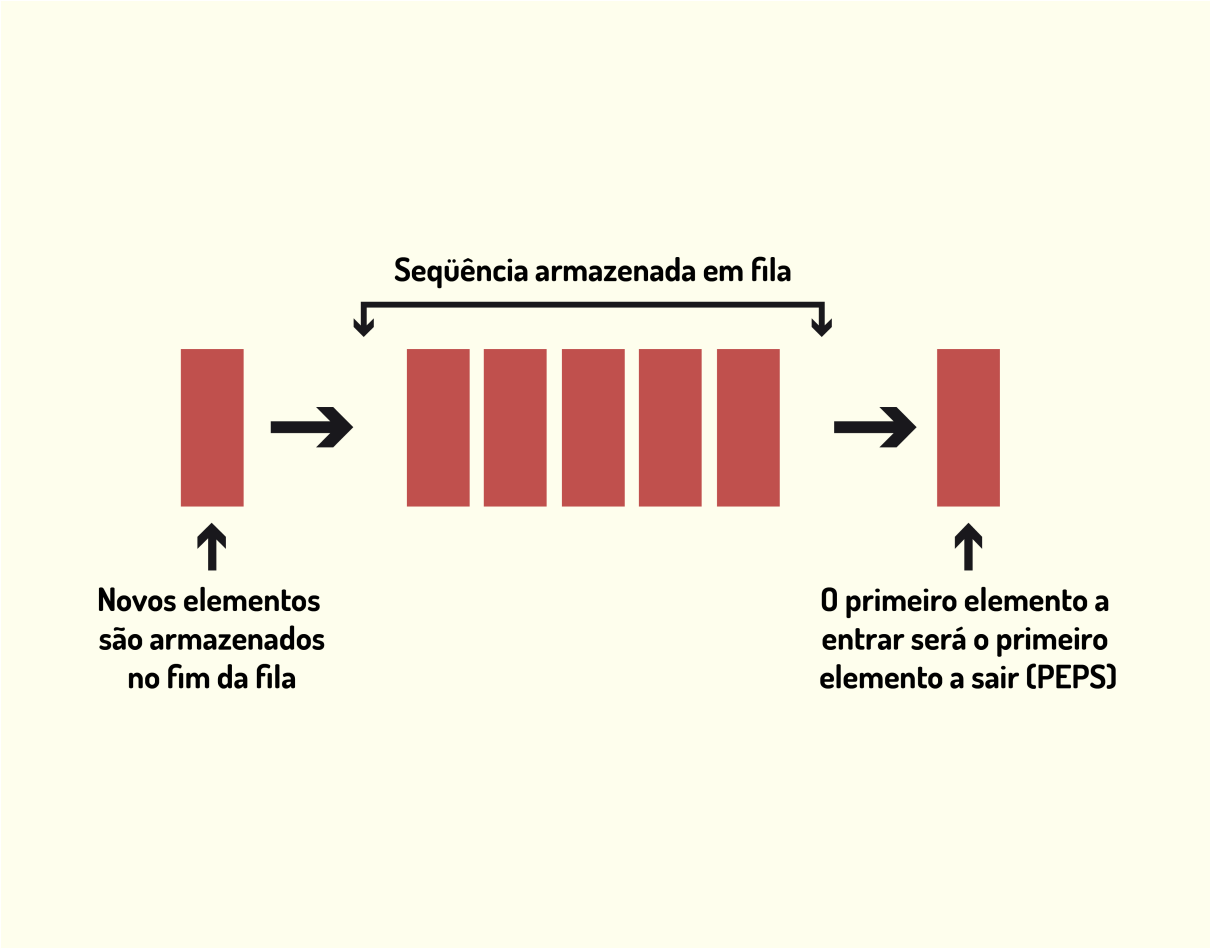
A busca em largura se utiliza de uma fila e também se inicia pelo nó raiz e desce para o próximo nó adjacente, a diferença é que olha todos os filhos do nó antes de descer na árvore.

Conceito de Filas

Enfrentamos filas em vários locais como em mercados, bancos, entradas de shows, entre outras situações. O conceito de uma fila no modo computacional é igual ao de uma fila convencional, em que esperamos para ser atendidos e, conforme a fila anda, vamos caminhando. Segundo Pulga e Rissetti (2009 p. 219):

> Conceito é conhecido como *first in, first out* ou FIFO, expressão conhecida em português como PEPS ou "primeiro que entra, primeiro que sai". Então, no conceito de fila, os elementos são atendidos, ou utilizados, sequencialmente na ordem em que são armazenados. As filas (*queues*) são conjuntos de elementos (as listas) cujas operações de inserção são feitas por extremidade, e as de remoção, por outra.

Como exemplo, pode-se utilizar o conceito da fila de um banco, em que as primeiras pessoas que chegam são as primeiras a serem atendidas.

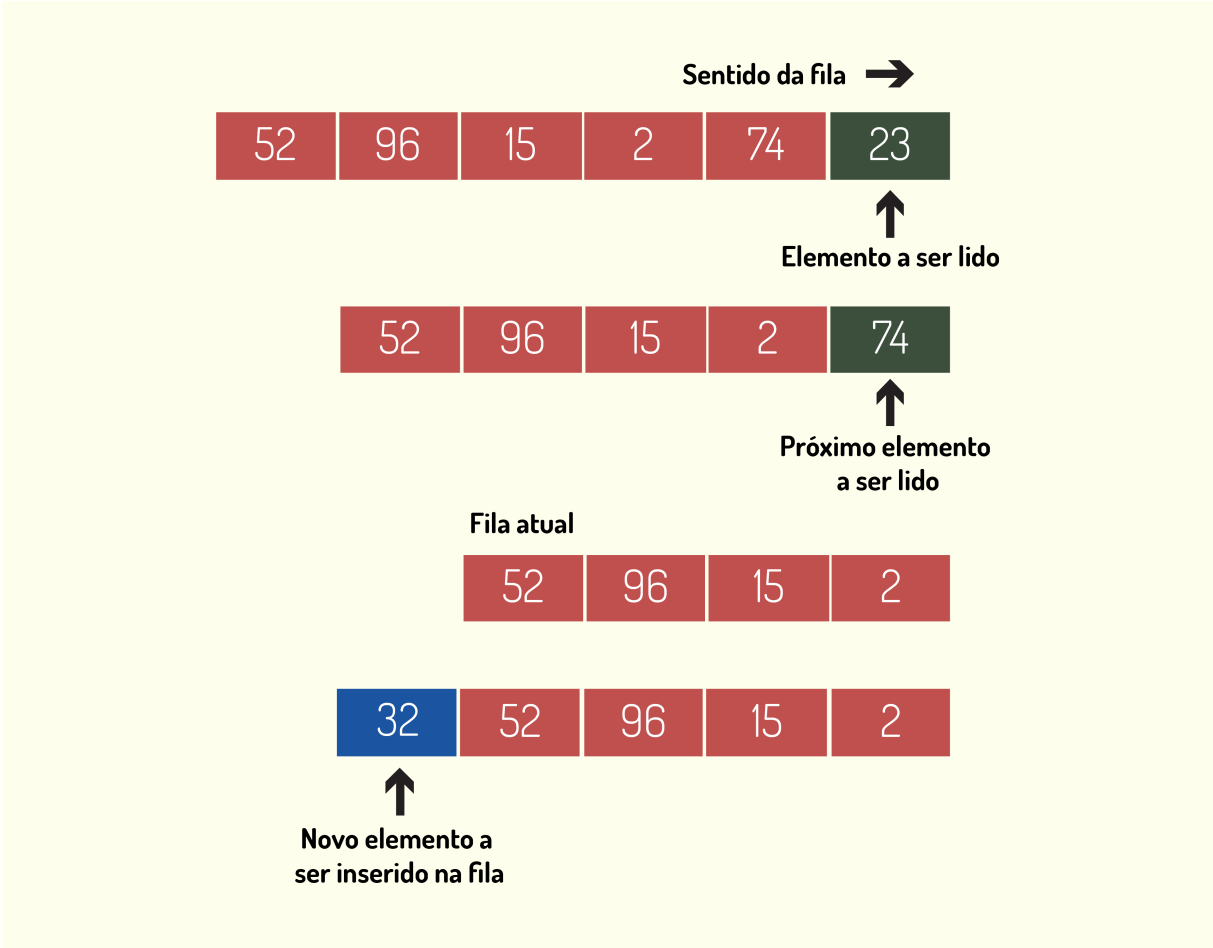


2FIGURA 15.18 - conceito de FIFO FONTE: Pulga e Rissetti (2009 p. 219).

Em uma lista linear, algumas das regras que devem ser seguidas nesse conceito de FIFO são:

- Os novos elementos sempre entram por último, no final da fila, mesmo se essa estiver ordenada.

- O elemento que é lido será o primeiro que entrou ou o próximo da lista.
- O tempo de espera na lista depende do programa em execução.
- A lista pode ficar vazia.



2FIGURA 16.18 - Exemplo de uma FILA em execução FONTE: o autor.

Implementação de Fila em C

Prezado(a) aluno(a), agora que você sabe como funciona uma fila de modo computacional, veja um código no qual foi implementado esse conceito, trata-se da Implementação da Estrutura FILA em linguagem C de Atilho (2013).

```
//FIFO
//Bibliotecas utilizadas
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

//Se o sistema for Windows adiciona determinada biblioteca, e definindo comandos de limpar e esperar
#ifdef WIN32
	#include <windows.h>
	#define LIMPA_TELA system(“cls”)
//Senão for Windows (se for Linux)
#else
	#include <unistd.h>
	#define LIMPA_TELA system(“/usr/bin/clear”)
#endif


//Estrutura da lista que será criada
typedef struct Fila {
	int valor;
	struct Fila *proximo;
} Dados;

void insere();
void exclui();
void mostra();
void mostra_erro();

//Inicializando os dados da lista
Dados *principal = NULL;
Dados *final = NULL;

main(){ setlocale(LC_ALL, “Portuguese”);
	char escolha;
	//Laço que irá mostrar o menu esperando uma opção (char)
	do {
		//Limpando a tela, e mostrando o menu
		LIMPA_TELA;
		printf(“\nMétodo Fila\n\n”);
		printf(“Escolha uma opção: \n”);
		printf(“\t1 - Inserir valor na Fila\n”);
		printf(“\t2 - Remover valor da Fila\n”);
		printf(“\t3 - Mostrar valores da Fila\n”);
		printf(“\t9 - Sair\n\n”);
		printf(“Resposta: “);
		scanf(“%c”, &escolha);
		switch(escolha) {
			//Inserindo
			case ‘1’:
			insere();
			break;
			//Excluindo
			case ‘2’:
			if(principal!=NULL){
			exclui();
			}
			else{
			printf(“\nA Fila está vazia!\n”);
				getchar();
			}
			break;
			//Mostrando
			case ‘3’:
			if(principal!=NULL){
				mostra();
			}
			else{
				printf(“\nA Fila está vazia!\n”);
				getchar();
			}
				break;
			case ‘9’:
			printf(“\nObrigado por utilizar esse programa!\n”);
			printf(“------>Terminal de Informação<------\n\n”);

			exit(0);
			break;
```

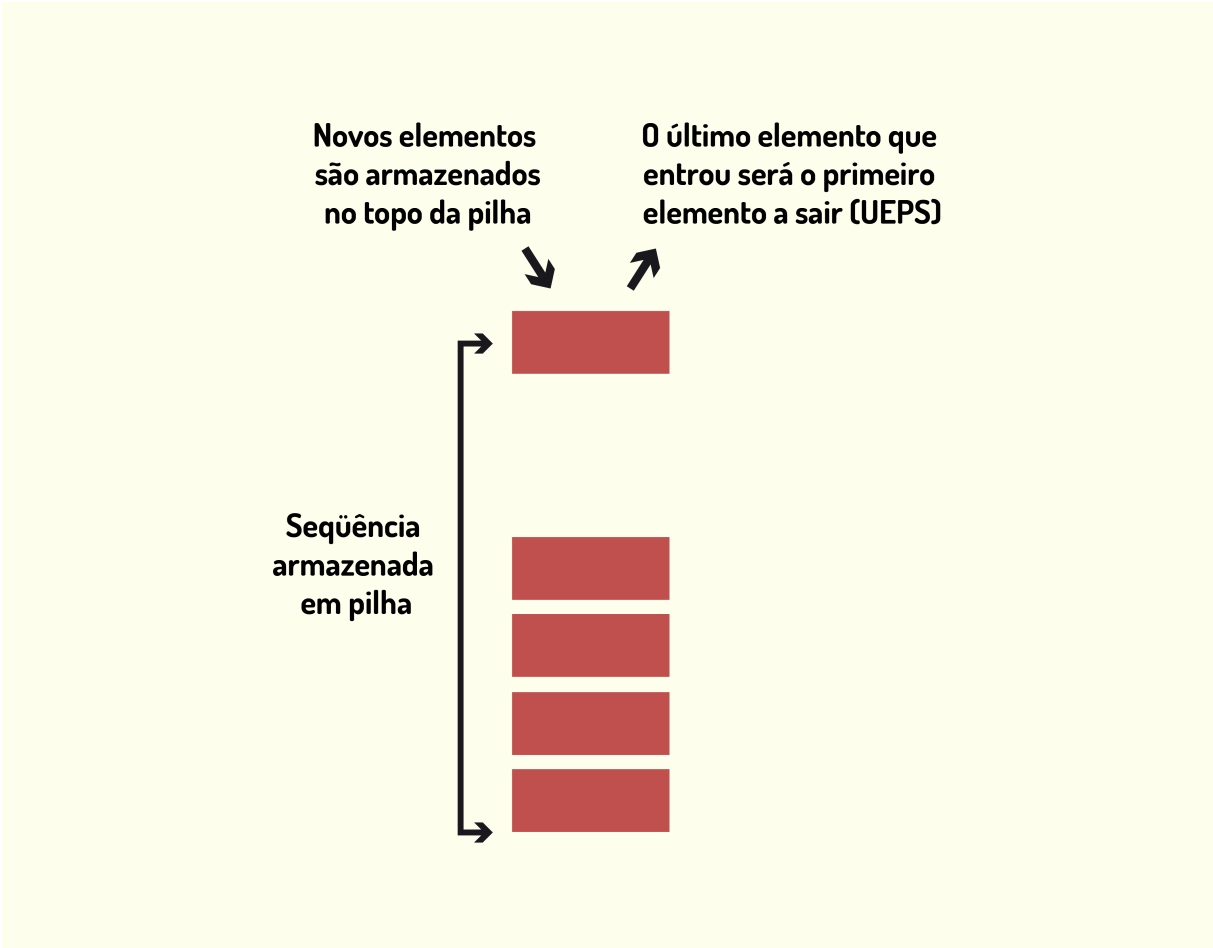
```
                        //Se foi algum valor inválido
                        default:
                                mostra_erro();
                                break;
                }
                //Impedindo sujeira na gravação da escolha
                getchar();
        }
        while (escolha > 0); //Loop Infinito
}

//Inserção
void insere(){
        int val;
        LIMPA_TELA;
        printf("\nInserção: \n");
        printf("---------------------------------------\n");
        printf("Insira o valor a ser inserido: ");
        scanf("%d",&val);
        Dados *atual = (Dados*)malloc(sizeof(Dados));
        atual -> valor = val;
        atual -> proximo = NULL;

        //se o principal estiver vazio, será o atual
        if(principal == NULL){
                principal = final = atual;
        }
        //senão, o próximo valor que será o atual
        else{
                final->proximo=atual;
                final=atual;
        }

        printf("\nValor inserido!\n");
        printf("---------------------------------------");
        getchar();
}

//Exclusão
void exclui(){
        Dados *auxiliar;
        printf("\nExclusão: \n");
        printf("---------------------------------------\n");
        //o auxiliar será o próximo da principal
        auxiliar=principal->proximo;
        //limpando a principal
        free(principal);
        //a principal será a auxiliar
        principal=auxiliar;
        printf("\nValor excluido!\n");
        printf("---------------------------------------");
        getchar();
}

//Mostrando
void mostra(){
        int posicao=0;
        Dados *nova=principal;
        LIMPA_TELA;
        printf("\nMostrando valores: \n");
        printf("---------------------------------------\n");
        //laço de repetição para mostrar os valores
        for (; nova != NULL; nova = nova->proximo) {
                posicao++;
                printf("Posição %d, contém o valor %d\n", posicao, nova->valor);
        }
        printf("---------------------------------------");
        getchar();
}

//Mostrando erro de digitação
void mostra_erro(){
        LIMPA_TELA;
        printf("\nErro de Digitação: \n");
        printf("---------------------------------------\n");
        printf("\nDigite uma opção válida (pressione -Enter- p/ continuar)!\n\n");
        printf("---------------------------------------");
```

```
        getchar();
}
```

Conceito de Pilhas

Em nosso cotidiano, nos deparamos com vários tipos de pilhas, como a pilha de pratos no armário da cozinha, pilha de roupas, com uma pilha de sacos de arroz no mercado. Uma estrutura em pilha é uma das mais simples no ambiente computacional e segue os mesmos princípios de uma pilha no mundo real.

> As pilhas são estruturas de dados conhecidas como lista LIFO (Last In, First Out); em português, significa que o último elemento a entrar é o primeiro a sair UEPS. Trata-se de uma lista linear em que todas as operações de inserção e remoção são feitas por um único extremo, denominado topo .
>
> (PUGA; RISSETTI, 2016 p.186)

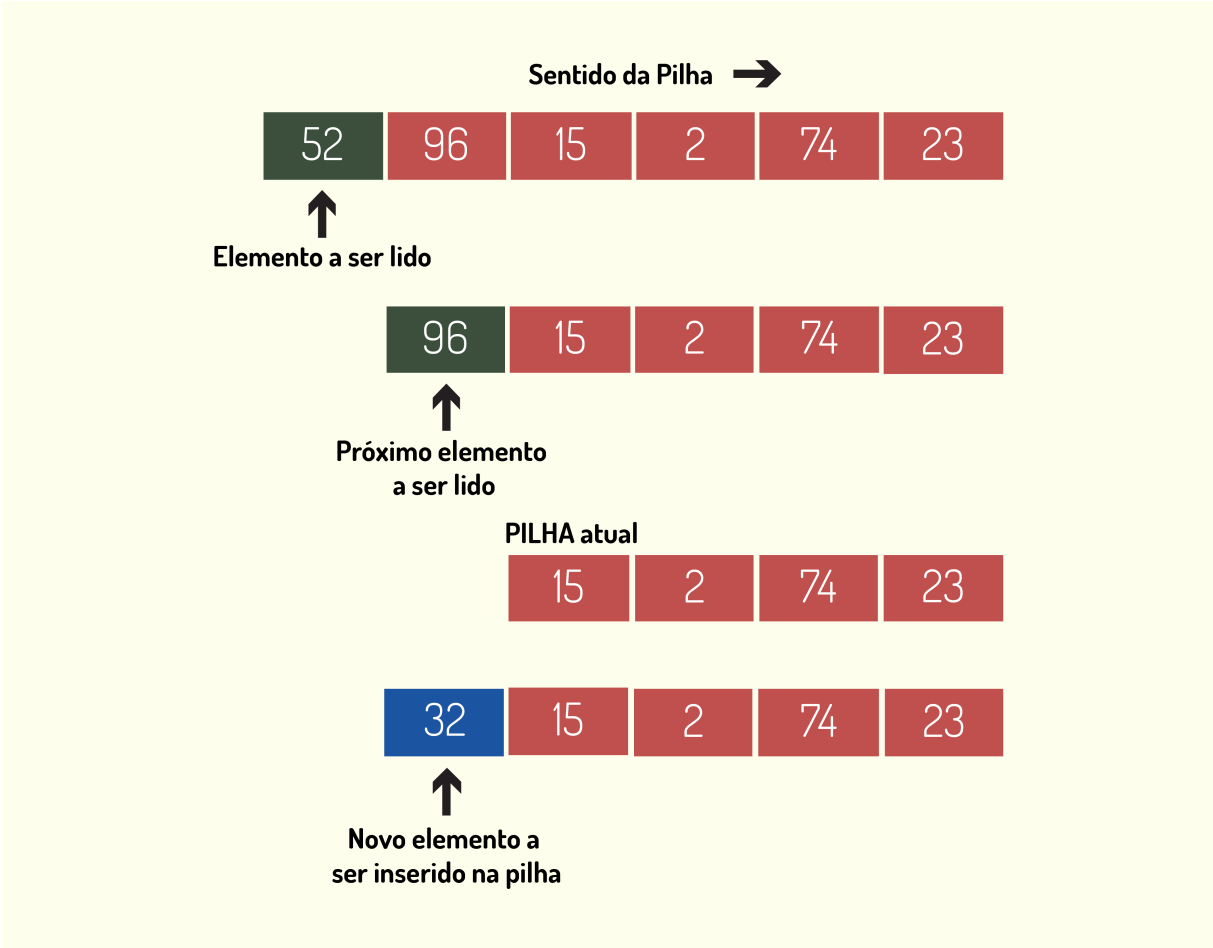


2FIGURA 17.18 - Conceito de pilha FONTE: Puga e Rissetti (2009 p.226).

Temos que ter cuidado com esse conceito de pilha na informática para não nos confundirmos, pois estaremos trabalhando com lista lineares, consequentemente, a sua representação não será visualmente igual a uma pilha, não sendo desenvolvida na vertical, mas sim na horizontal.

Deve-se respeitar os seguintes pontos:

- Os novos elementos sempre entram por último, no final da pilha, mesmo se a essa estiver ordenada.
- O elemento que é lido será o último inserido ou o próximo da pilha.
- O tempo de espera na pilha depende do programa em execução.
- A pilha pode ficar vazia.



2FIGURA 18.18 - exemplo de uma pilha em execução

Caro(a) aluno(a), veja que não podemos ler o primeiro elemento que foi inserido até que todos os demais acima dele sejam lidos, nesse tipo de estrutura, o primeiro elemento inserido será o último. Sendo assim, lembre-se de que "os últimos elementos inseridos serão os primeiros".

Implementação de Pilha em C

Prezado(a) aluno(a), agora que você sabe como funciona uma pilha de modo computacional, veja um código no qual foi implementado esse conceito, tratata de uma implementação de uma pilha em linguagem C de Atilho (2013).

```
//LIFO
//Bibliotecas utilizadas
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

//Se o sistema for Windows adiciona determinada biblioteca, e definindo comandos de limpar e esperar
#ifdef WIN32
	#include <windows.h>
	#define LIMPA_TELA system(“cls”)
//Senão for Windows (se for Linux)
#else
	#include <unistd.h>
	#define LIMPA_TELA system(“/usr/bin/clear”)
#endif

//Estrutura da lista que será criada
typedef struct pilha {
	int valor;
	struct pilha *proximo;
} Dados;

void insere();
void exclui();
void mostra();
void mostra_erro();

//Inicializando os dados da lista
Dados *principal = NULL;

main(){ setlocale(LC_ALL, “Portuguese”);
	char escolha;
	//Laço que irá mostrar o menu esperando uma opção (char)
	do {
		//Limpando a tela, e mostrando o menu
		LIMPA_TELA;
		printf(“\nMétodo Pilha\n\n”);
		printf(“Escolha uma opção: \n”);
		printf(“\t1 - Inserir valor na Pilha\n”);
		printf(“\t2 - Remover valor da Pilha\n”);
		printf(“\t3 - Mostrar valores da Pilha\n”);
		printf(“\t9 - Sair\n\n”);
		printf(“Resposta: “);
		scanf(“%c”, &escolha);
		switch(escolha) {
			//Inserindo
			case ‘1’:
				insere();
				break;
			//Excluindo
			case ‘2’:
				if(principal!=NULL){
				exclui();
			}
			else{
				printf(“\nA Pilha está vazia!\n”);
				getchar();
			}
			break;
			//Mostrando
			case ‘3’:
			if(principal!=NULL){
				mostra();
			}
			else{
				printf(“\nA Pilha está vazia!\n”);
				getchar();
			}
			break;
			case ‘9’:
			printf(“\nObrigado por utilizar esse programa!\n”);
			printf(“------>Terminal de Informação<------\n\n”);
				exit(0);
			break;
			//Se foi algum valor inválido
			default:
			mostra_erro();
```

```
                        break;
                }
                //Impedindo sujeira na gravação da escolha
                getchar();
        }
        while (escolha > 0); //Loop Infinito
}

//Inserção
void insere(){
        int val;
        LIMPA_TELA;
        printf("\nInserção: \n");
        printf("--------------------------------------\n");
        printf("Insira o valor a ser inserido: ");
        scanf("%d",&val);
        //gerando a posição atual
        Dados *atual = (Dados*)malloc(sizeof(Dados));
        atual -> valor = val;
        //próximo do atual será a principal
        atual -> proximo = principal;
        //principal volta a ser a atual
        principal = atual;
        printf("\nValor inserido!\n");
        printf("--------------------------------------");
        getchar();
}

//Exclusão
void exclui(){
        Dados *auxiliar;
        printf("\nExclusão: \n");
        printf("--------------------------------------\n");
        //guardando o valor depois da principal
        auxiliar=principal->proximo;
        //limpando a principal
        free(principal);
        //a principal será a auxiliar
        principal=auxiliar;
        printf("\nValor excluido!\n");
        printf("--------------------------------------");
        getchar();
}

//Mostrando
void mostra(){
        int posicao=0;
        Dados *nova=principal;
        LIMPA_TELA;
        printf("\nMostrando valores: \n");
        printf("--------------------------------------\n");
        //laço de repetição para mostrar os dados
        for (; nova != NULL; nova = nova->proximo) {
                posicao++;
                printf("Posição %d, contém o valor %d\n", posicao, nova->valor);
        }
        printf("--------------------------------------");
        getchar();
}

//Mostra erro de digitação
void mostra_erro(){
        LIMPA_TELA;
        printf("\nErro de Digitação: \n");
        printf("--------------------------------------\n");
        printf("\nDigite uma opção válida (pressione -Enter- p/ continuar)!\n\n");
        printf("--------------------------------------");
        getchar();
}
```

Indicação de leitura

Nome do livro: Estruturas de Dados Usando C

Editora: Pearson MAKRON

Autor: Tenenbaum, Aaron; Langsam, Yedidyah; Augenstein, Moshe J.

ISBN: 8534603480

Ajudando os leitores a formar estruturas de dados eficientes em C, esse livro explica como aplicar as estruturas de dados para otimizar a execução de programas. Com forte ênfase sobre o projeto estruturado e as técnicas de programação, ele apresenta informações precisas sobre todos os passos envolvidos no desenvolvimento de estruturas de dados, desde a concepção teórica até a realização concreta. Inclui diversas implementações alternativas de estruturas de dados e conselhos sobre a escolha da estrutura mais adequada às suas necessidades imediatas; inúmeros exemplos de programação; desenvolvimento completo de todos os programas; algoritmos de classificação e de busca; descobertas de pesquisas recentes.

UNIDADE III

Técnicas de Ordenação

Carlos Danilo Luz

Caro(a) aluno(a), seja bem-vindo(a) a esta unidade. Agora, aprenderemos os conceitos sobre ordenação de dados e quais as técnicas utilizadas para efetuar esse processo. Ao manipular informações, quanto menos movimentos ou procedimentos forem executados para encontrar o que se busca, melhor. Serão compreendidos alguns dos algoritmos de ordenação tais como o Bubble Sort, Insertion Sort, Shell Sort, entre outros. Estudaremos os métodos como cada um desses algoritmos se aplica à busca, será visto que alguns algoritmos terão melhor performance sobre outros dependendo do vetor ou do método aplicado e que alguns deles foram criados para melhorar outros já existentes. Ao final de cada tópico, será vista uma implementação dos conceitos em linguagem C.

Técnicas de Ordenação

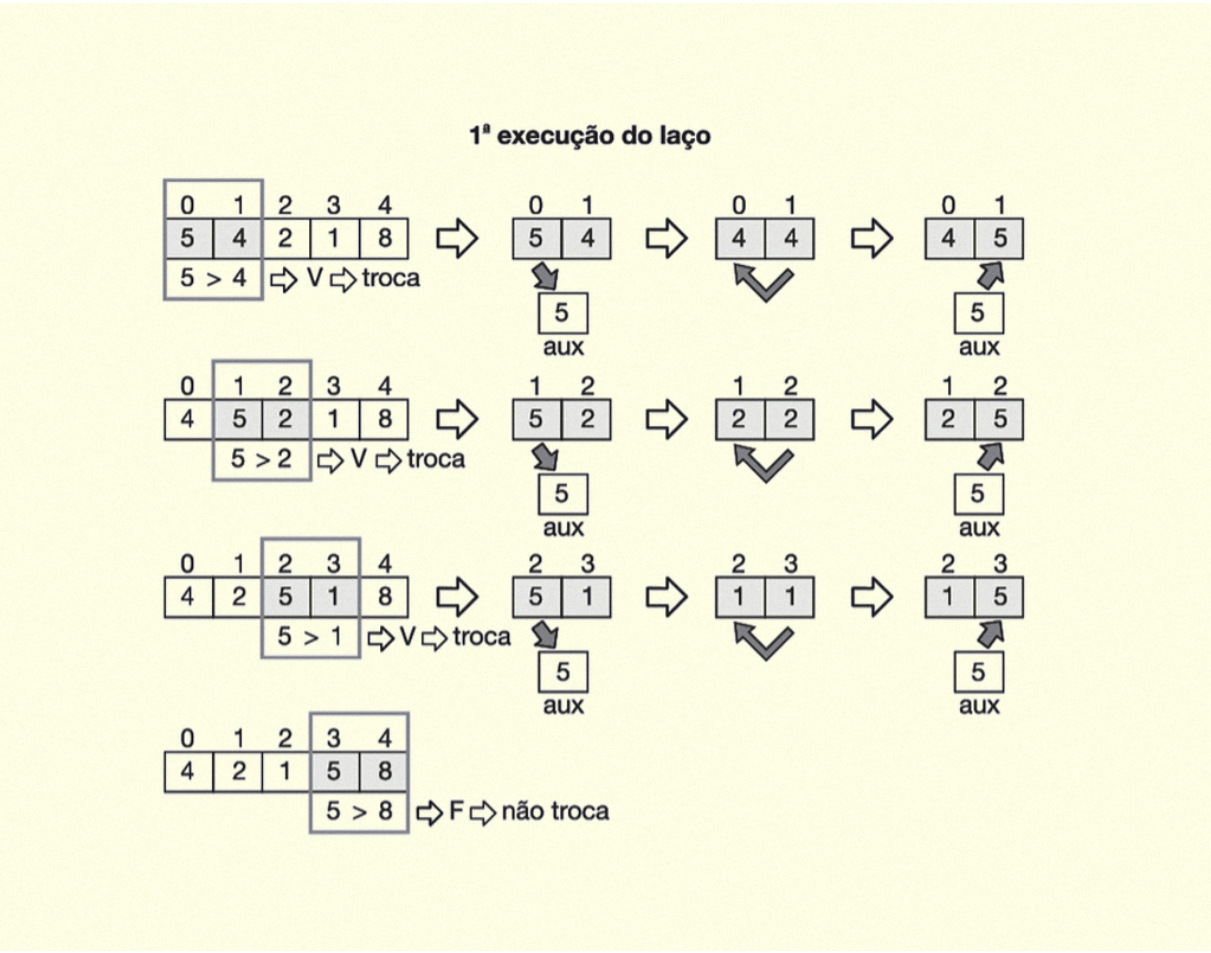
O que mais se busca no ambiente computacional é uma melhora constante de desempenho, um dos itens que impactam diretamente nisso é a forma que os dados estão disponíveis. As técnicas e algoritmos de ordenação são de grande importância nesse momento e são baseados no uso de índices ou em listas ordenadas. Você irá compreender como os algoritmos de ordenação executam esse procedimento.

BubbleSort

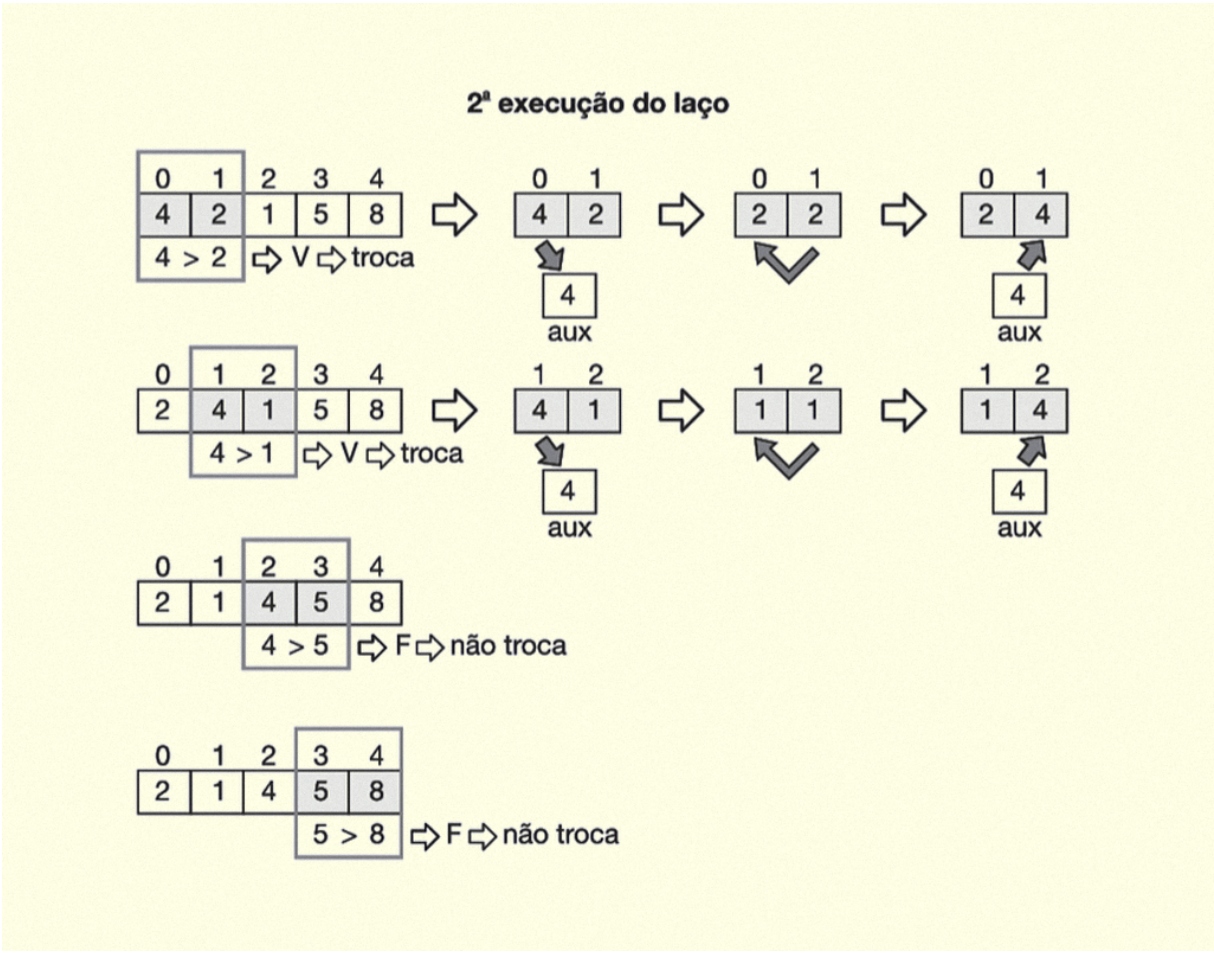
O método de ordenação BubbleSort, também conhecido como método da bolha, consiste em efetuar a troca dos elementos vizinhos no vetor, seu modo de execução consiste em:

> Realizar a ordenação comparando os elementos, dois a dois, e trocando-se de posição, de acordo com o critério estabelecido de ordem crescente ou decrescente. Seu nome se deve à ideia de que os elementos vão "subindo" para as posições corretas, como bolhas .
>
> (PUGA; RISSETTI, 2016 p.172)

A ordenação é efetuada por meio de listas lineares, vamos utilizar como exemplo um vetor com 5 posições, no qual queremos efetuar a ordenação crescente dos valores:



3FIGURA 1.29 - 1º laço de ordenação FONTE: Ascencio e Araújo (2010 p.22).

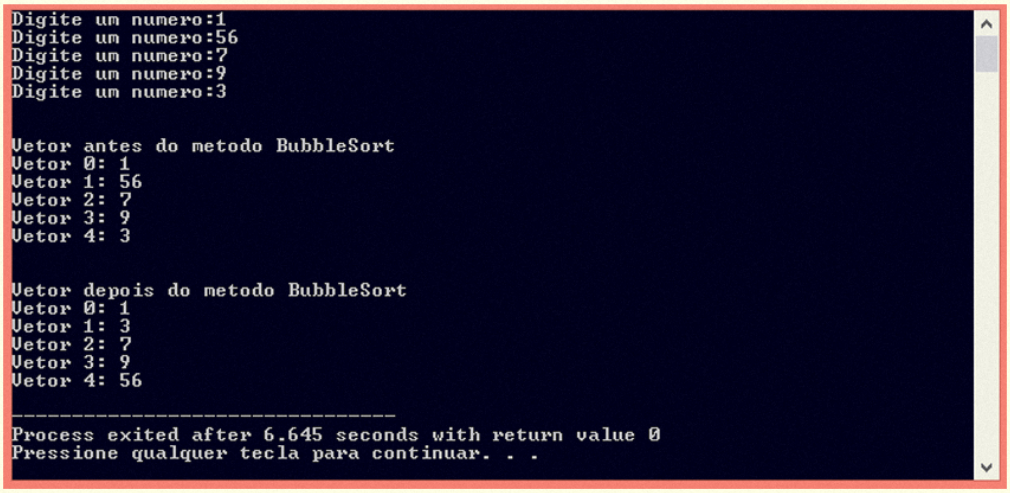


3FIGURA 2.29 - 2º laço de ordenação FONTE: Ascencio e Araújo (2010 p.22).

A quantidade de laços de repetição não é definida com base no exemplo apresentado, a lista não estava totalmente ordenada no final do 2° laço, nesse caso, o algoritmo e rodado novamente, até que toda a lista seja ordenada, o maior problema desse método é que ele deve percorrer todo os elementos, comparando dois a dois, verificando se o elemento está em ordem ou não, em listas menores, isso não será um grande problema, contudo, em lista maiores, a complexidade irá aumentar, pois será necessário um número maior de interações.

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

main(){

    int X[5], n, i, aux;

    // inserindo os números no vetor
    for(i=0; i<=4; i++){
        printf("Digite um numero:");
        scanf("%d", &X[i]);
    }
    printf("\n\nVetor antes do metodo BubbleSort\n");
    for(i=0;i<=4;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
    // ordenando de forma crescente
    for(n=1;n<=5;n++){
        // laço que percorre e ordena o vetor
        for(i=0;i<=4;i++){
            if(X[i]>X[i+1]){
                aux = X[i];
                X[i]=X[i+1];
                X[i+1]=aux;
            }
        }
    }

    printf("\n\nVetor depois do metodo BubbleSort\n");
    // monstrando o vetor ordenado
    for(i=0;i<=4;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
}
```

3FIGURA 3.29 - Script em linguagem C, BubbleSort FONTE: DEV C++



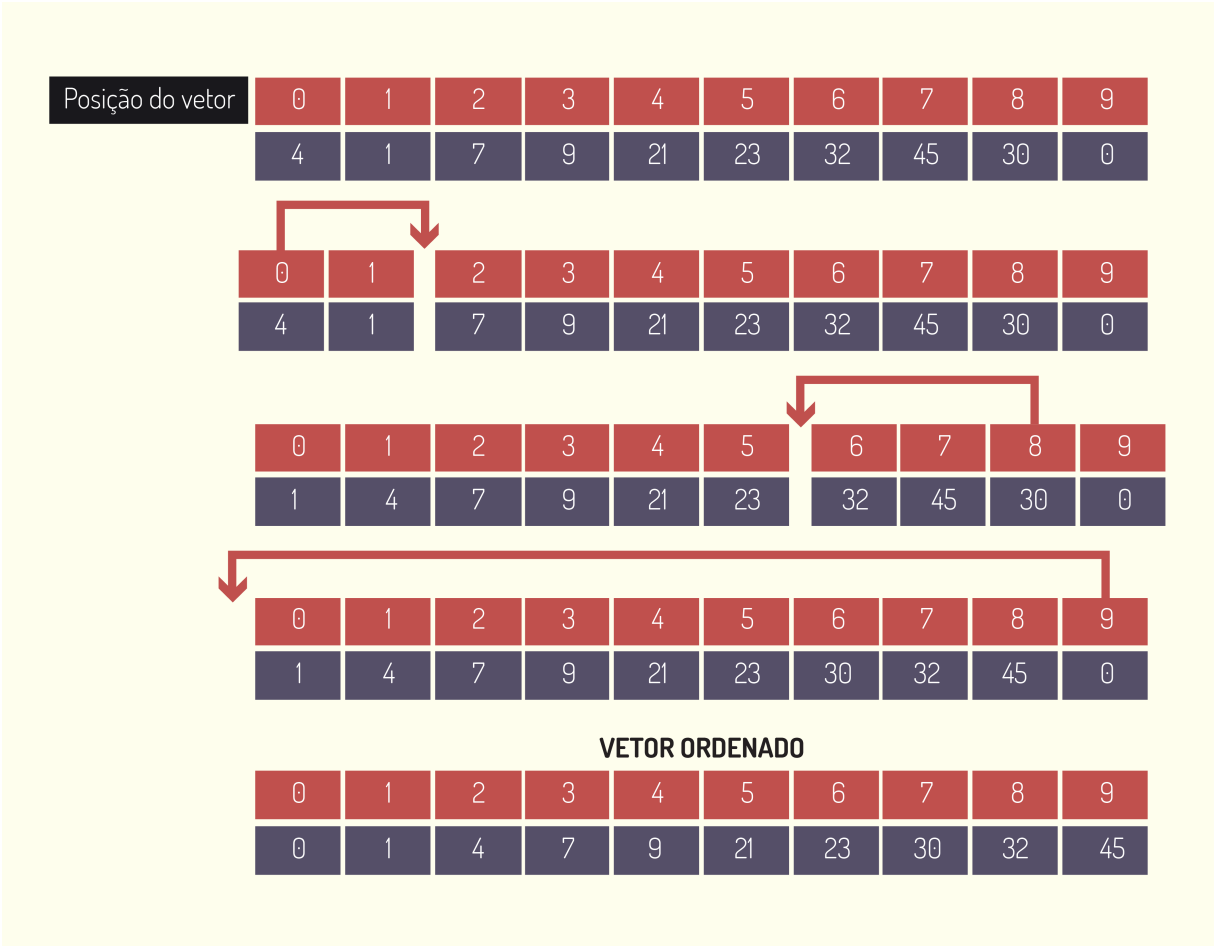
3FIGURA 4.29 - Resultado do programa BubbleSort FONTE: BubbleSort

InsertionSort

O método de ordenação InsertionSort também é conhecido como inserção de elemento para ordenação, ele traz resultados bons através de uma implementação simples e consiste no seguinte:

> Será eleito o segundo número do vetor para iniciar a ordenação; assim, os elementos à esquerda do número eleito estão sempre ordenados de forma crescente ou decrescente. Logo, um laço com as comparações será executado do segundo elemento ao último, ou seja, na quantidade de vezes igual ao número de elementos do vetor menos um.[...] Enquanto existirem elementos à esquerda do número eleito para comparações e a posição que atende a ordenação que se busca não for encontrada, o laço será executado .
>
> (ASCENCIO; ARAÚJO 2010 p.38)

Ao executar o InsertionSort, o primeiro elemento do vetor é eleito e removido do vetor para efetuar as comparações com os demais, ao encontrar sua posição ideal, o elemento eleito é reinserido no vetor e assim um novo elemento é eleito, repetindo o processo até que esteja tudo ordenado. Vejamos o exemplo a seguir:

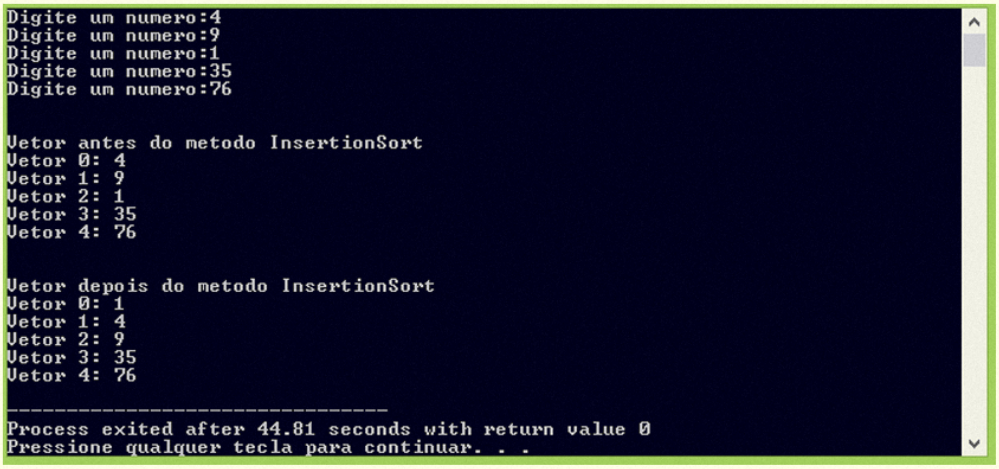


3FIGURA 5.29 - Implementação do método de InsertionSort FONTE: Elaborada pelo autor.

Ao utilizar o método InsertionSort, temos menos interações para ordenação do vetor, após selecionar um elemento, o vetor é percorrido, sendo possível reinseri-lo à esquerda ou à direita.

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

main(){

    int X[5], j, i, insert;

    // inserindo os números no vetor
    for(i=0; i<=4; i++){
        printf("Digite um numero:");
        scanf("%d", &X[i]);
    }
    printf("\n\nVetor antes do metodo InsertionSort\n");
    for(i=0;i<=4;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
    // ordenando de forma crescente
    for(i=1;i<=5;i++){
        insert = X[i];
        j=i-1;
        // laço que percorre até encontrar a
        // posição para realocação do número eleito
        while(j>=0 && X[j] > insert){
            X[j+1] = X[j];
            j = j - 1;
        }
        X[j+1] = insert;
    }

    printf("\n\nVetor depois do metodo InsertionSort\n");
    // monstrando o vetor ordenado
    for(i=0;i<=4;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
}
```

3FIGURA 6.29 - Script em linguagem C, InsertionSort FONTE: DEV C++



3FIGURA 7.29 - Execução do script InsertionSort FONTE: Script InsertionSort

Reflita

O SelectionSort compara a posição atual de um elemento com os demais elementos do vetor, mesmo se o valor atual for menor ou maior, ele irá percorrer todo o vetor. Como o método prever percorrer todo o vetor, se o vetor não for ordenado ou parcialmente ordenado, o tempo de execução computacional será o mesmo.

Técnicas de Ordenação

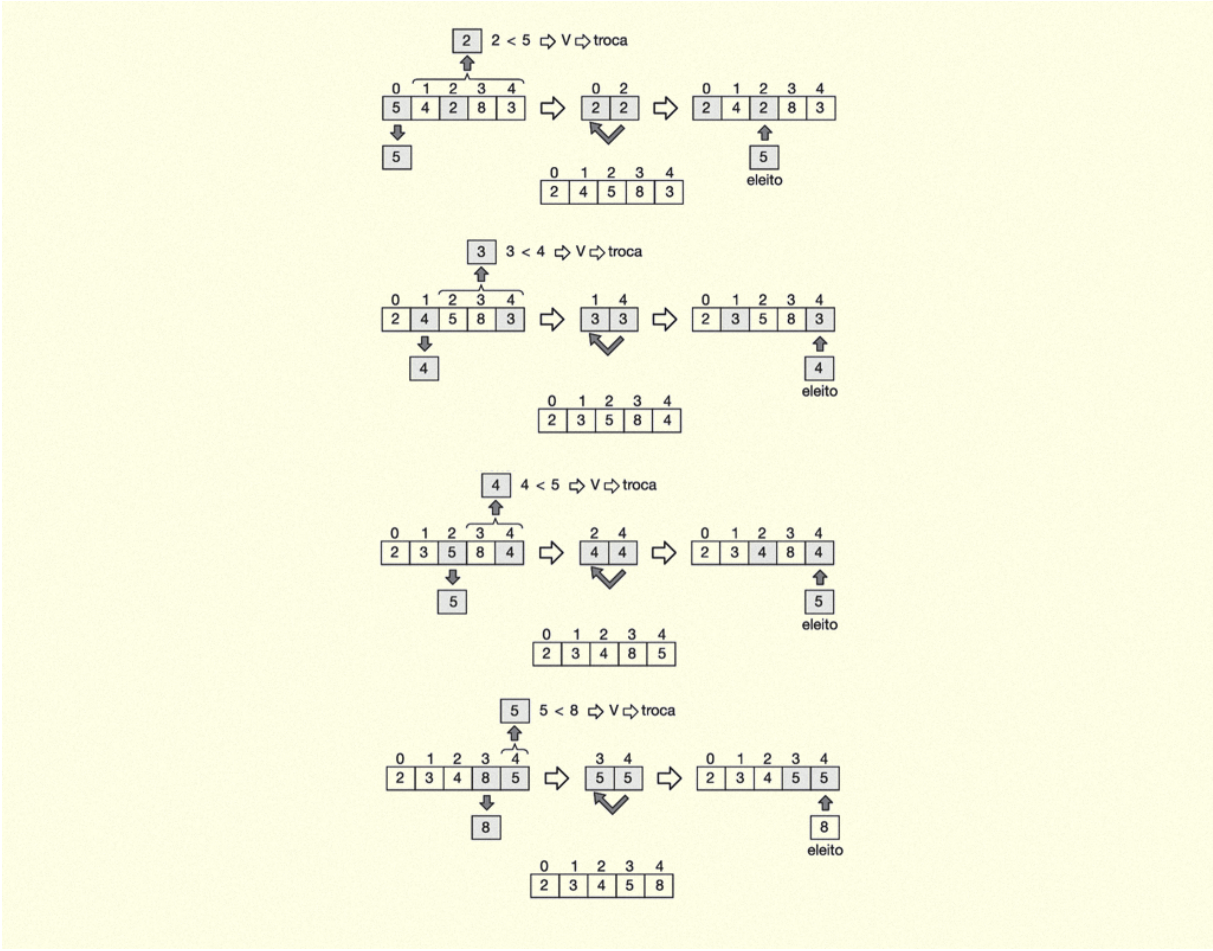
Na unidade anterior, estudamos dois métodos de ordenação e busca, ambos de simples entendimento. Veremos agora novos métodos com uma complexidade maior que os já apresentados.

SelectionSort

O método é bem parecido com o InsertionSort e também bem simples. Ele consiste em trocar os elementos de posição, ao se escolher um elemento, ele percorre todo o vetor e efetuando as trocas quando necessário.

> Neste algoritmo de ordenação cada número do vetor, a partir do primeiro, será eleito e comparado com o menor ou maior, dependendo da ordenação desejada, número dentre aqueles que estão à direita do eleito. Nessas comparações procura-se um número menor que o eleito (quando a ordenação for crescente) ou um maior que o eleito (quando a ordenação for decrescente). Quando um número satisfaz as condições da ordenação desejada (menor ou maior que o eleito), este trocará de posição com o eleito, assim, todos os números à esquerda do eleito ficam sempre ordenados .
>
> (ASCENCIO; ARAÚJO 2010 p.44)

A cada nova interação, é efetuada uma busca à direita do elemento escolhido e verificado se esse elemento é o menor do que o que estamos comparando, caso este elemento seja menor é efetuada a troca entre os dois, o eleito e novo elemento, vejamos como o método funciona com um vetor de 5 elementos (5, 4, 2, 8, 3).

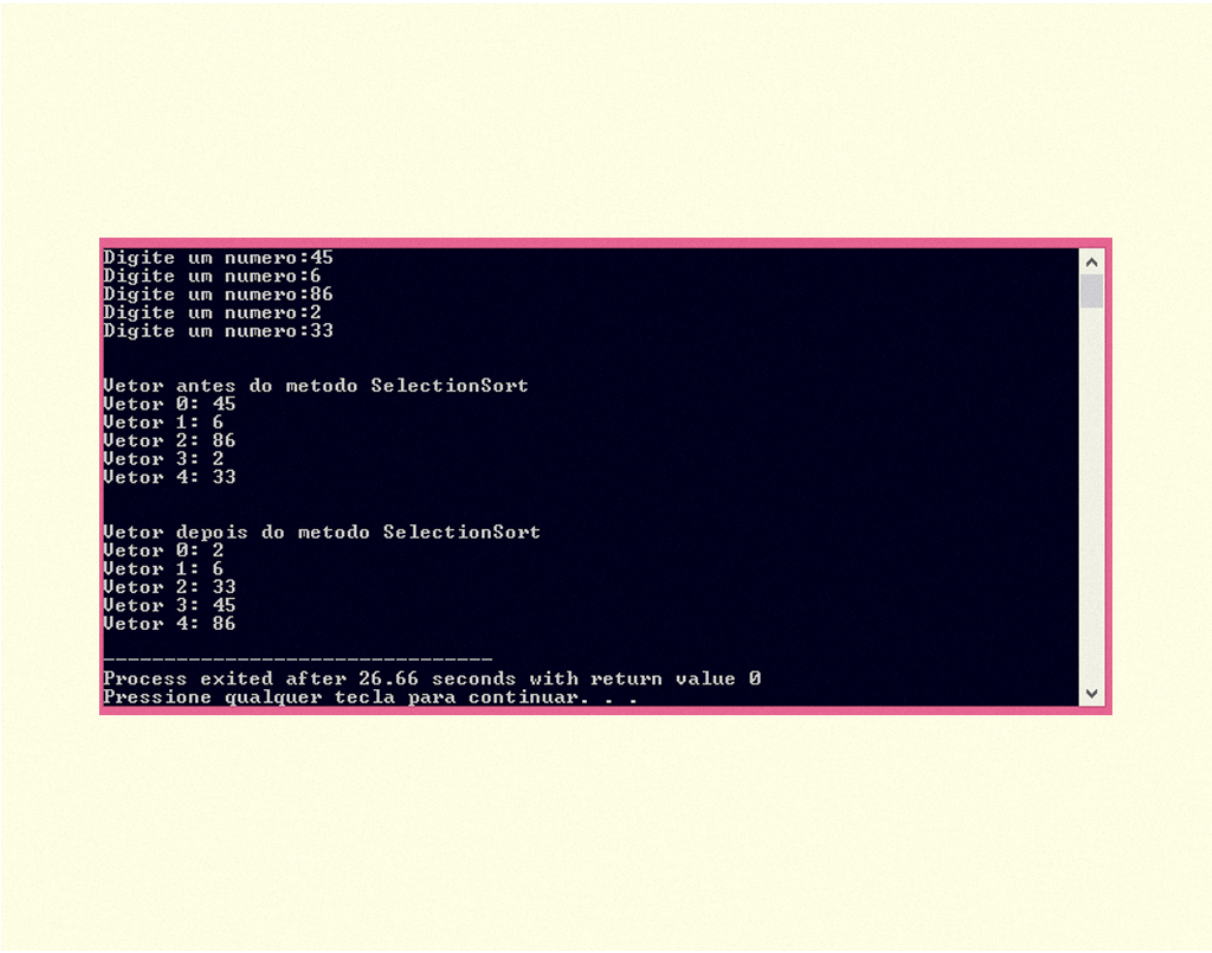


3FIGURA 8.29 - Implementação do método SelectionSort FONTE: adaptada de Ascencio e Araújo (2010).

Perceba, prezado(a) aluno(a), que o método do selectionsort é bem parecido com o conceito do bubblesort, pois ele compara valores, dois a dois, para depois efetuar a troca, a diferença é que o selectionsort busca o menor número do resto do vetor à direita, já o método bubblesort compara os valores com o seu vizinho da direita, efetuando a troca de lugares caso este elemento for menor.

```c
#include <stdio.h>
#include <conio.h>
main(){
    int X[5], j, i, select, menor, troc;
    // inserindo os números no vetor
    for(i=0; i<=4; i++){
        printf("Digite um numero:");
        scanf("%d", &X[i]);
    }
    printf("\n\nVetor antes do metodo SelectionSort\n");
    for(i=0;i<=4;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
    // ordenando de forma crescente
    for(i=0;i<=3;i++){
        select = X[i];
        menor = X[i+1];
        troc = i+1;

        for(j=i+1;j<=4;j++){
            if(X[j] < menor) {
                menor = X[j];
                troc=j;
            }
        }
        if(menor < select){
            X[i]=X[troc];
            X[troc] = select;
        }
    }
    printf("\n\nVetor depois do metodo SelectionSort\n");
    // monstrando o vetor ordenado
    for(i=0;i<=4;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
}
```

3FIGURA 9.29 - Script em linguagem C, SelectionSort FONTE: DEV C++.



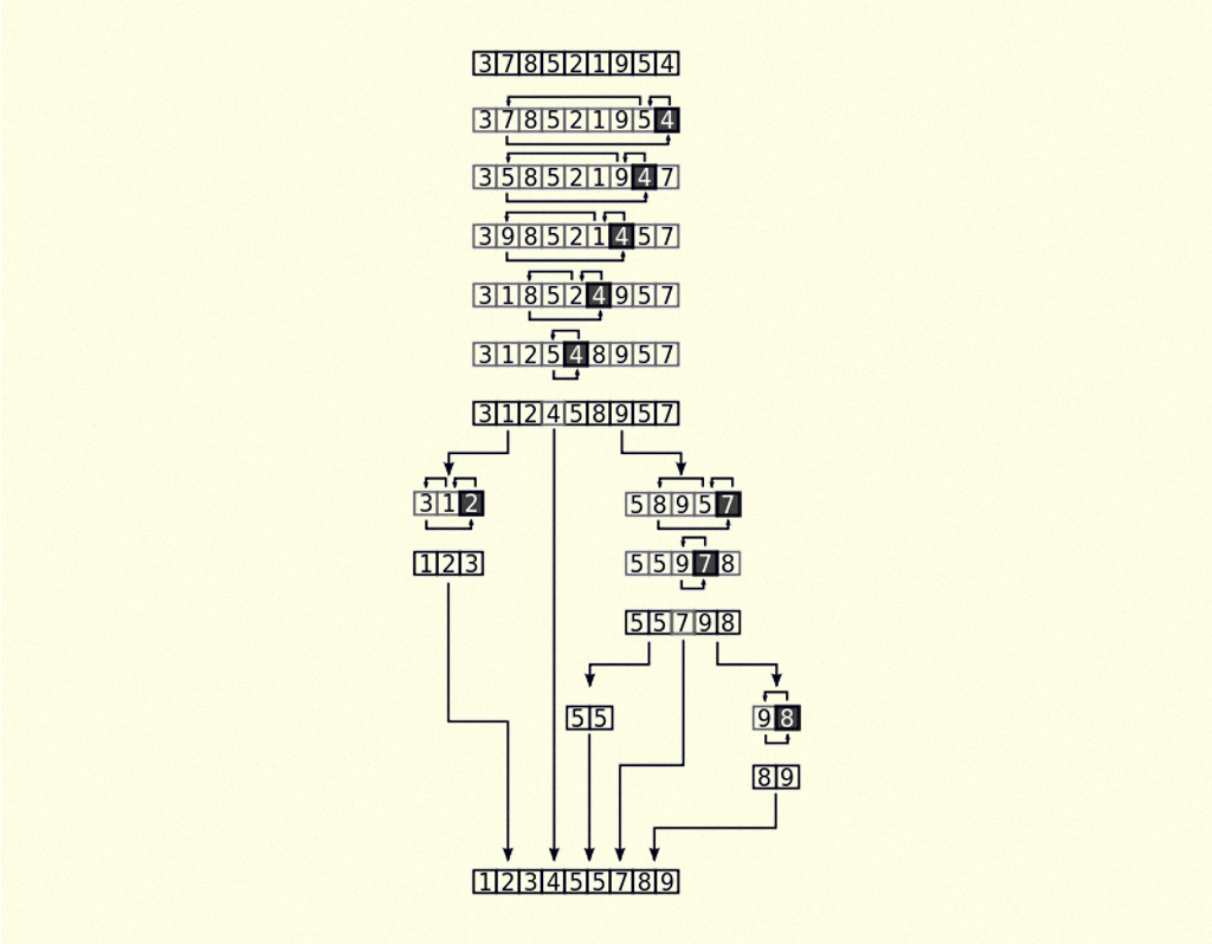
3FIGURA 10.29 - resultado da execução do script de selectionsort FONTE: Selectionsort.

QuickSort

O método Quicksort é considerado o algoritmo de ordenação mais utilizado no mundo. Foi criado em 1960 pelo cientista britânico Sir Charles Antony Richard Hoare e publicado em 1962 com várias melhorias. Segundo Cormem (2012), o método também é conhecido por classificação por troca de partição. Tem o conceito de dividir o vetor para ordenar.

> Neste algoritmo de ordenação, o vetor é particionado em dois por meio de um procedimento recursivo. Essa divisão ocorre até que o vetor fique com apenas um elemento, enquanto os demais ficam ordenados à medida que ocorre o particionamento .
>
> (ASCENCIO; ARAÚJO 2010 p.44)

O método desse algoritmo consiste em escolher um elemento qualquer e denominá-lo pivô, a partir desse ponto o vetor é dividido em três sublistas, uma com os valores menores que o vetor, outra com os valores maiores e a terceira com o próprio pivô. Nas sublistas não ordenadas, é escolhido um novo pivô e efetuada a divisão novamente, esse processo é realizado até que a lista seja completamente ordenada. Vejamos a seguir um exemplo desse método na ordenação de 10 elementos, o elemento escolhido como pivô inicial será o número 4, a partir dele será efetuada a ordenação do vetor.

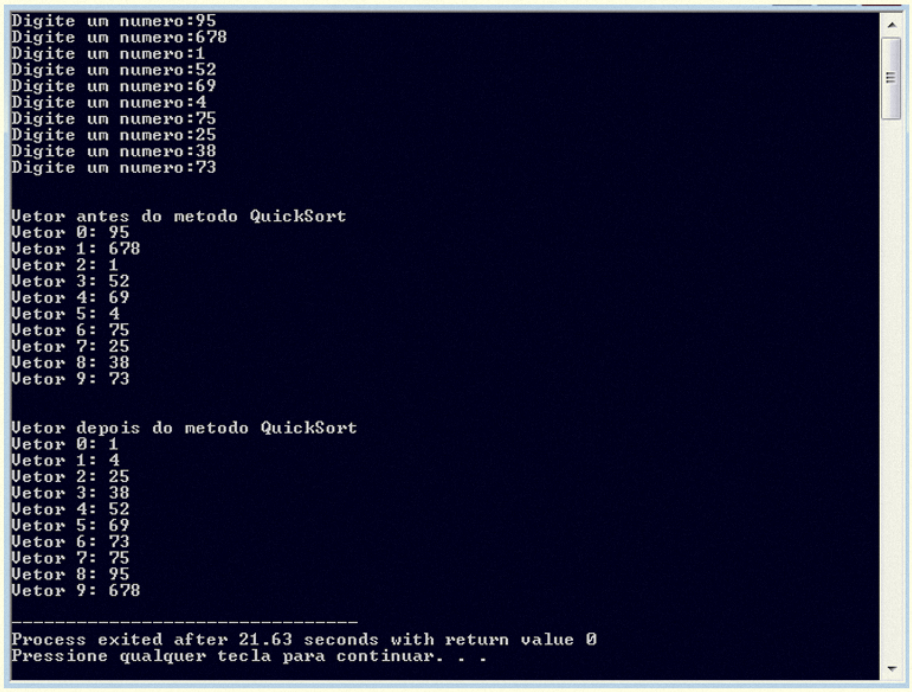


3FIGURA 11.29 - Implementação do método QuickSort FONTE: Aplicação... (2009).

O algoritmo vai separando os elementos até que todo o vetor fique ordenado, posteriormente, ele vai reinserindo as sublistas no vetor de forma já ordenada. Esse algoritmo tem uma complexidade maior por conta da separação, entretanto é mais eficaz do que os já apresentados, nos quais é efetuada a comparação sempre em dois elementos do vetor, caso o elemento de comparação seja menor se efetua a troca, nesses métodos, a lista sempre vai ser lida por completo várias vezes. No método QuickSort, o vetor ou lista já irá ser separada forma ordenada. Vejamos o exemplo desse método em linguagem C.

```c
#include <stdio.h>
#include <conio.h>
    void quick_sort (int *vet, int total) {
        int i, j, p, t;
        if (total < 2)
            return;
        p = vet[total / 2];
        for (i = 0, j = total - 1;; i++, j--) {
            while (vet[i] < p)
                i++;
            while (p < vet[j])
                j--;
            if (i >= j)
                break;
            t = vet[i];
            vet[i] = vet[j];
            vet[j] = t;
        }
        quick_sort(vet, i);
        quick_sort(vet + i, total - i);
    }
main(){
    int X[10], i;
    // inserindo os números no vetor
    for(i=0; i<=9; i++){
        printf("Digite um numero:");
        scanf("%d", &X[i]);
    }
    printf("\n\nVetor antes do metodo QuickSort\n");
    for(i=0;i<=9;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
    // ordenando de forma crescente

    quick_sort(X, 10);

    printf("\n\nVetor depois do metodo QuickSort\n");
    // monstrando o vetor ordenado
    for(i=0;i<=9;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
}
```

3FIGURA 12.29 - Script em linguagem C, QuickSort FONTE: DEV C++.



3FIGURA 13.29 - Resultado do programa QuickSort FONTE: QuickSort.

Técnicas de Ordenação

Veremos agora novos tipos de ordenação e busca, com base nos já foram apresentados até o momento.

ShellSort

O algoritmo de Shellsort foi desenvolvido por Donald Shell e publicado em 1959 pela Universidade de Cincinnat. O nome do método não tem relação com uma concha (shell, em inglês), sendo uma homenagem ao seu criador. Esse algoritmo se utiliza da ordenação por inserção.

> O método shellsort pode ser considerado um refinamento do método de inserção direta, diferindo pelo fato de no lugar de considerar o vetor a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles. Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores .
>
> (OLIVEIRA; PRADA; SILVA, 2002)

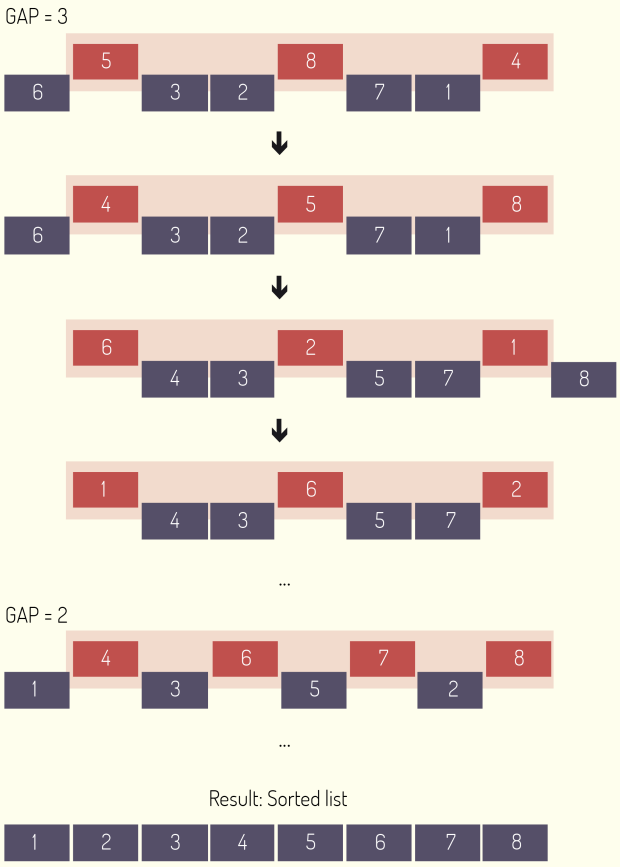
Podemos definir esse método como a junção do QuickSort e do InsertionSort, pois ele divide o vetor em sublistas depois aplica a busca e inserção de elementos na mesma.

A implementação possui uma variável chamada de gap. Esse GAP determina a distância entre os elementos que são separados do vetor original, sendo aplicado o insertionsort, após ordenada a sublista, esta é reinserida no vetor original. O valor do GAP é decrementado conforme as sublistas são criadas e aplicadas ao método do insertionsort, o processo se repete até o gap atingir o valor igual a 1, efetuando uma busca simples igual ao bubblesort.

Imagine um vetor com 8 posições, e o gap inicial sendo 3, a primeira sublista se inicia da posição 0 do vetor, essa irá conter os seguintes elementos: shell[0], shell[3], shell[6].

Após efetuar essa separação, teremos uma sublista com 3 elementos no qual aplicamos o insertionsort, depois retornamos para o vetor original. É efetuada uma nova separação agora a partir da posição 1 do vetor, contendo os elementos: shell[1], shell[4], shell[7].

São criadas novas sublistas até que todos os elementos sejam contemplados finalizando a primeira passagem, após isso, o gap é decrementado em -1, sendo, então 2. Será efetuado o procedimento até que o vetor esteja ordenado.

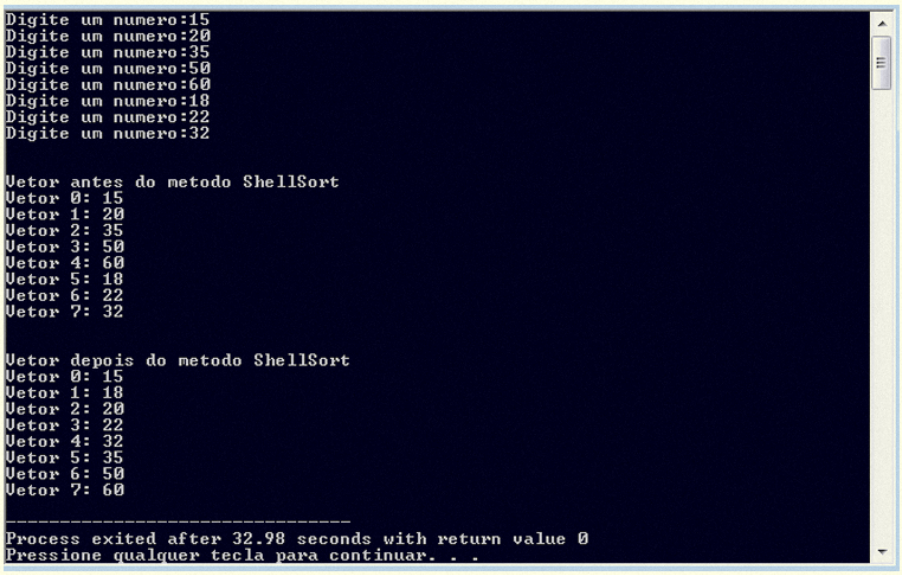


3FIGURA 14.29 - Método ShellSort FONTE: Stoimen's web log.

Caro(a) aluno(a), vejamos a seguir um exemplo do método shellsort utilizando a linguagem C.

```c
#include <stdio.h>
#include <conio.h>
main(){
    int X[8], j, i, value, gap = 1;
    // inserindo os números no vetor

    for(i=0; i<=7; i++){
        printf("Digite um numero:");
        scanf("%d", &X[i]);
    }
    printf("\n\nVetor antes do metodo ShellSort\n");
    for(i=0;i<=7;i++){
        printf("Vetor %d: %d\n", i, X[i]);
    }
    // ordenando de forma crescente
        while(gap < 8) {
            gap = 3*gap+1;
        }
        while ( gap > 1) {
            gap /= 3;
            for(i = gap; i <=7; i++) {
                value = X[i];
                j = i - gap;
                while (j >= 0 && value < X[j]) {
                    X[j + gap] = X[j];
                    j -= gap;
                }
                X[j + gap] = value;
            }
        }
    printf("\n\nVetor depois do metodo ShellSort\n");
    // mostrando o vetor ordenado
    for(i=0;i<=7;i++){
        printf("Vetor %d: %d\n", i, X[i]);
    }
}
```

3FIGURA 15.29 - Script em linguagem C, ShellSort FONTE: DEV C++.



3FIGURA 16.29 - Resultado da execução do script em linguagem C FONTE: QuickSort.

¶ Fique por dentro

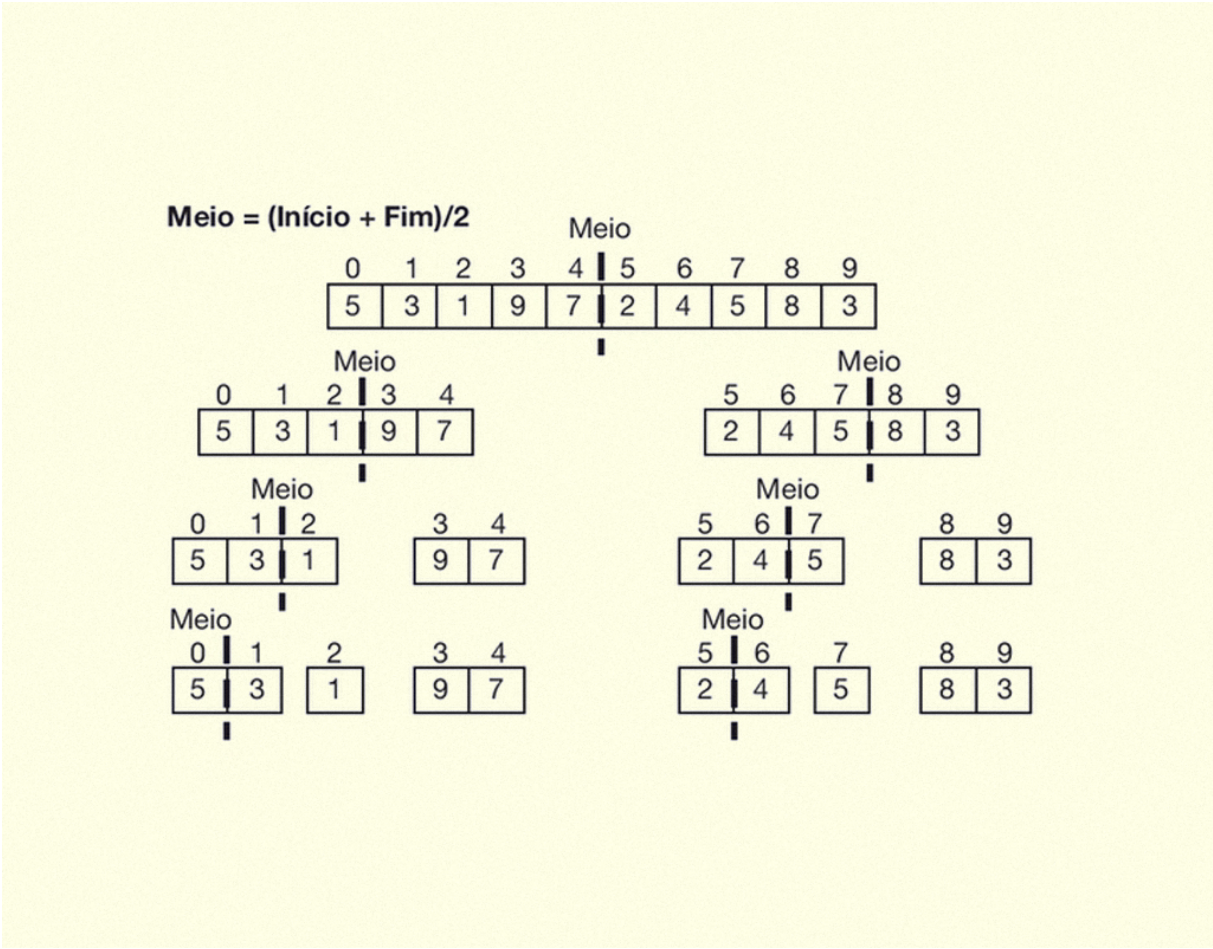
Caro(a) aluno(a), caso tenha curiosidade de ver como cada algoritmo de ordenação funciona, indico o link: **www.youtube.com. <https://www.youtube.com/watch?v=PbuuRLD_tg4>** Esse vídeo é do professor Rogério de Leon Pereira, que traz, de uma forma bem simples e visual, como cada um dos tipos de ordenação aqui apresentados funciona.

MergeSort

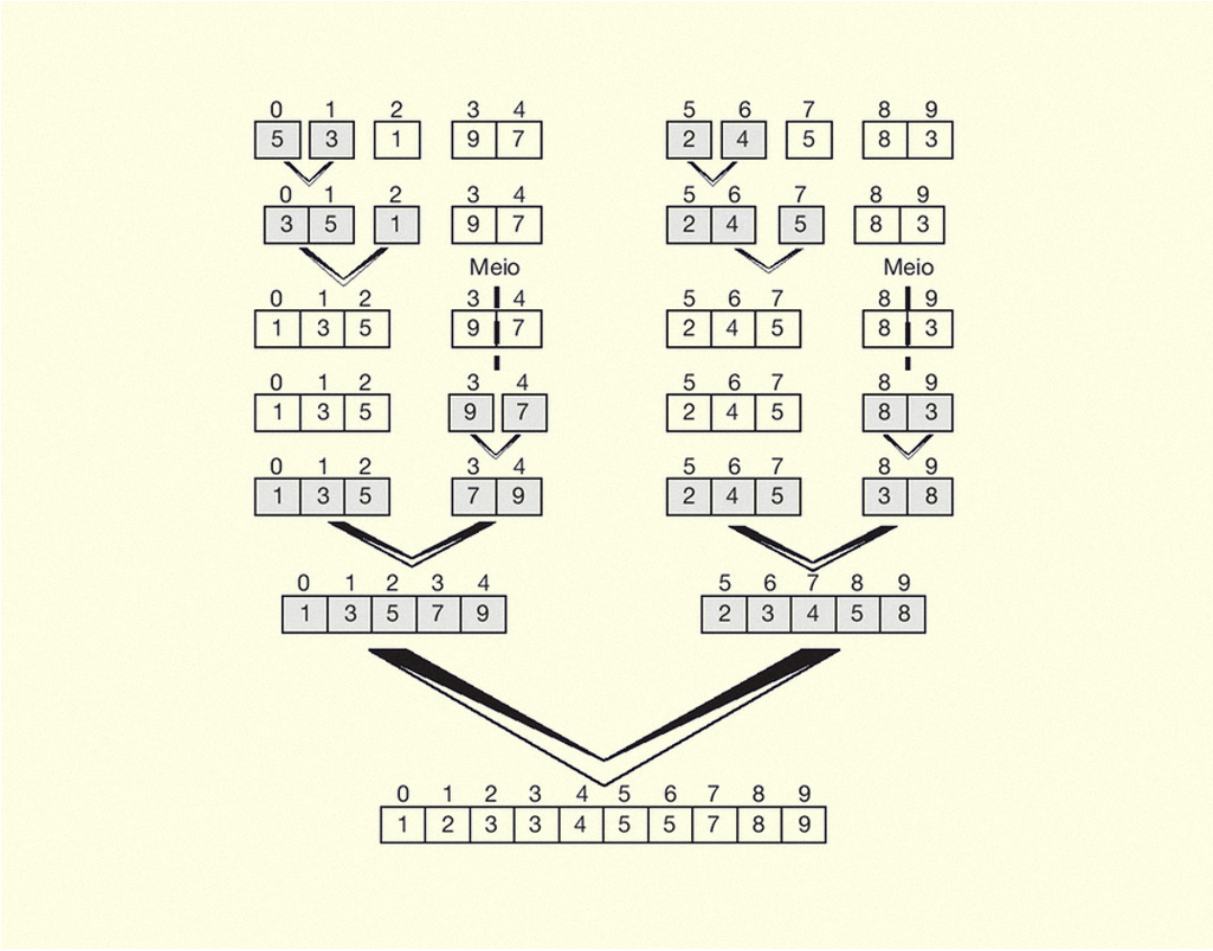
O método MergeSort também é conhecido como um algoritmo que divide para conquistar. Ele se utiliza do conceito de que um grande problema pode ser dividido em outros dois menores, e esses dois podem ser divididos em outros quatro menores ainda e assim por diante, deixando o vetor da forma mais simples possível. Após isso, reagrupam-se novamente essas partes

em ordem até que todas estejam reagrupadas.

A técnica faz essa divisão de forma recursiva, após efetuar o processo uma vez, ela é aplicada novamente até que fique apenas um único valor separado, que, em seguida, é comparado e já ordenado. Mesmo o MergeSort sendo mais complexo do que os demais métodos já apresentados, o esforço computacional é menor, porém, para cada nova divisão, é criado um novo vetor dinamicamente na memória, ao trabalharmos com vetores muitos grandes, isso pode consumir muito espaço de alocação.



3FIGURA 17.29 - 1º parte da ordenação, divisão do vetor FONTE: Ascencio e Araújo (2010 p.53).



3FIGURA 18.29 - 2º parte da ordenação, comparação e reagrupamento em ordem FONTE: Ascencio e Araújo (2010 p.54).

O algoritmo de MergeSort é considerado um evolução do BublleSort e do SelectionSort, mesmo sendo dividido até o nível mais baixo, ele é bem parecido com o QuickSort, pois ambos se utilizam do conceito dividir para conquistar. Com base em Ascencio e Araújo (2010), apresentaremos o Script em linguagem C, MergeSort.

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

void intercala(int X[], int inicio, int fim, int meio){
        int posLivre,inicio_vetor1, inicio_vetor2, i;
        int aux[10];
        inicio_vetor1 = inicio;
        inicio_vetor2 = meio+1;
        posLivre = inicio;
        while(inicio_vetor1 <= meio && inicio_vetor2 <= fim){
                if (X[inicio_vetor1] <= X[inicio_vetor2])
                {
                        aux[posLivre] = X[inicio_vetor1];
                        inicio_vetor1++;
                }
                else{
                        aux[posLivre] = X[inicio_vetor2];
                        inicio_vetor2++;
                }
                posLivre++;
        }
        //se ainda existem numeros no primeiro vetor
        //que nao foram intercalados
        for (int i = inicio_vetor1; i <= meio; ++i)
        {
                aux[posLivre] = X[i];
                posLivre++;
        }
        //se ainda existem numeros no segundo vetor
        //que nao foram intercalados
        for (int i = inicio_vetor2; i <= fim; ++i)
        {
                aux[posLivre] = X[i];
                posLivre++;
        }
        //retorne os valores do vetor aux para o vetor X
        for (int i = inicio; i <=fim; ++i)
        {
                X[i] = aux[i];
        }
}
void merge (int X[], int inicio, int fim){
        int meio;
        if (inicio < fim)
        {
                meio = (inicio+fim)/2;
                merge (X,inicio, meio);
                merge (X,meio+1, fim);
                intercala(X,inicio, fim, meio);

        }
}

main(){

int X[8], i;

for(i=0; i<=7; i++){
         printf(“Digite um numero:”);
         scanf(“%d”, &X[i]);
}
printf(“\n\nVetor antes do metodo MergeSort\n”);
for(i=0;i<=7;i++){
         printf(“Vetor %d: %d\n”, i, X[i]);
}
// ordenando de forma crescente
        merge(X,0,7);

printf(“\n\nVetor depois do metodo MergeSort\n”);
// mostrando o vetor ordenado
for(i=0;i<=7;i++){
         printf(“Vetor %d: %d\n”, i, X[i]);
}
}
```

3FIGURA 19.29 - Resultado da execução do script em linguagem C FONTE: QuickSort.

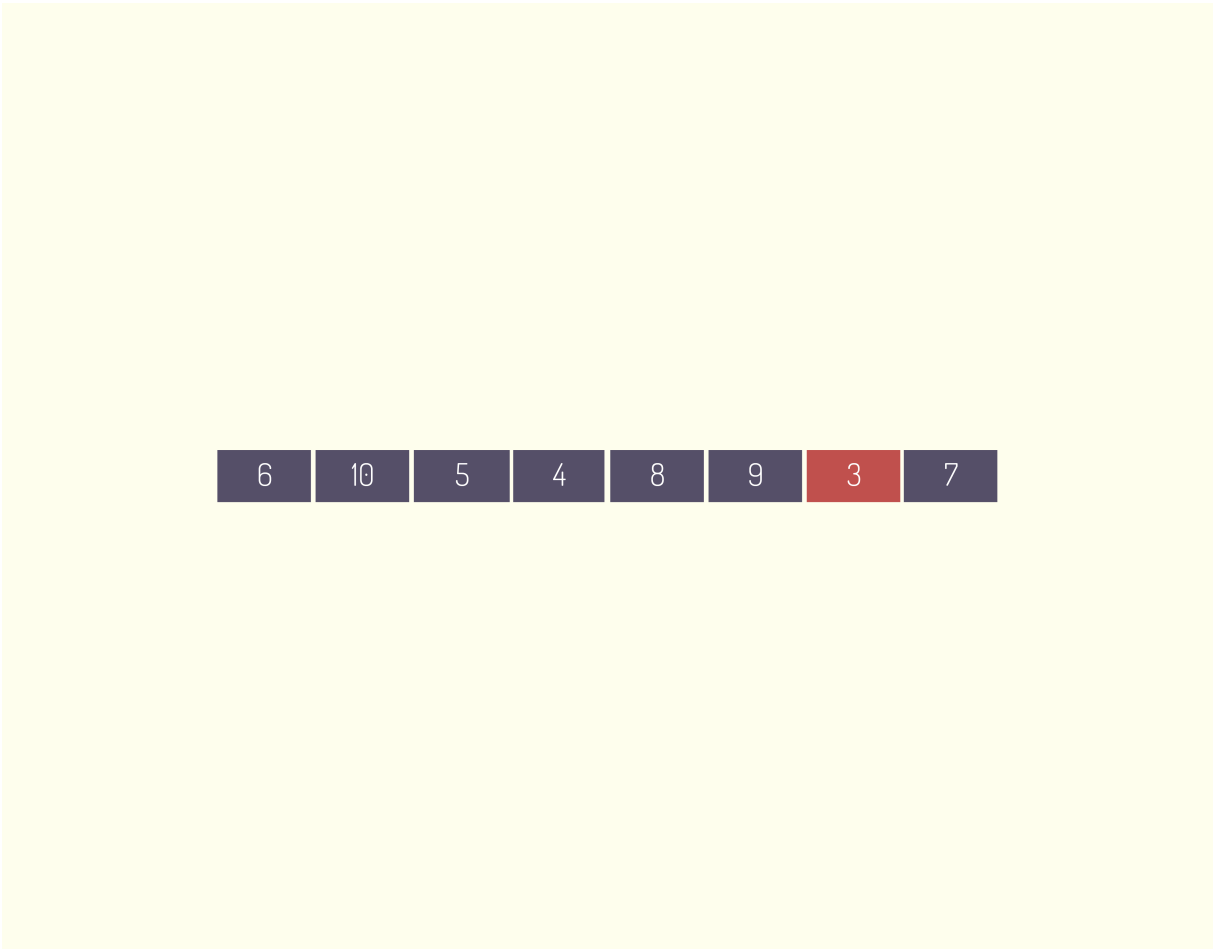
Técnicas de Ordenação

Vamos conhecer agora duas novas técnicas que propõem melhorias nos algoritmos já estudados.

CombSort

O algoritmo de CombSort busca uma melhoria no BubbleSort, o mesmo se passa na troca de dois elementos com base na comparação, o objetivo do CombSort é eliminar os *turtles* (tartarugas no inglês), mas o que seriam esses *turtles*?

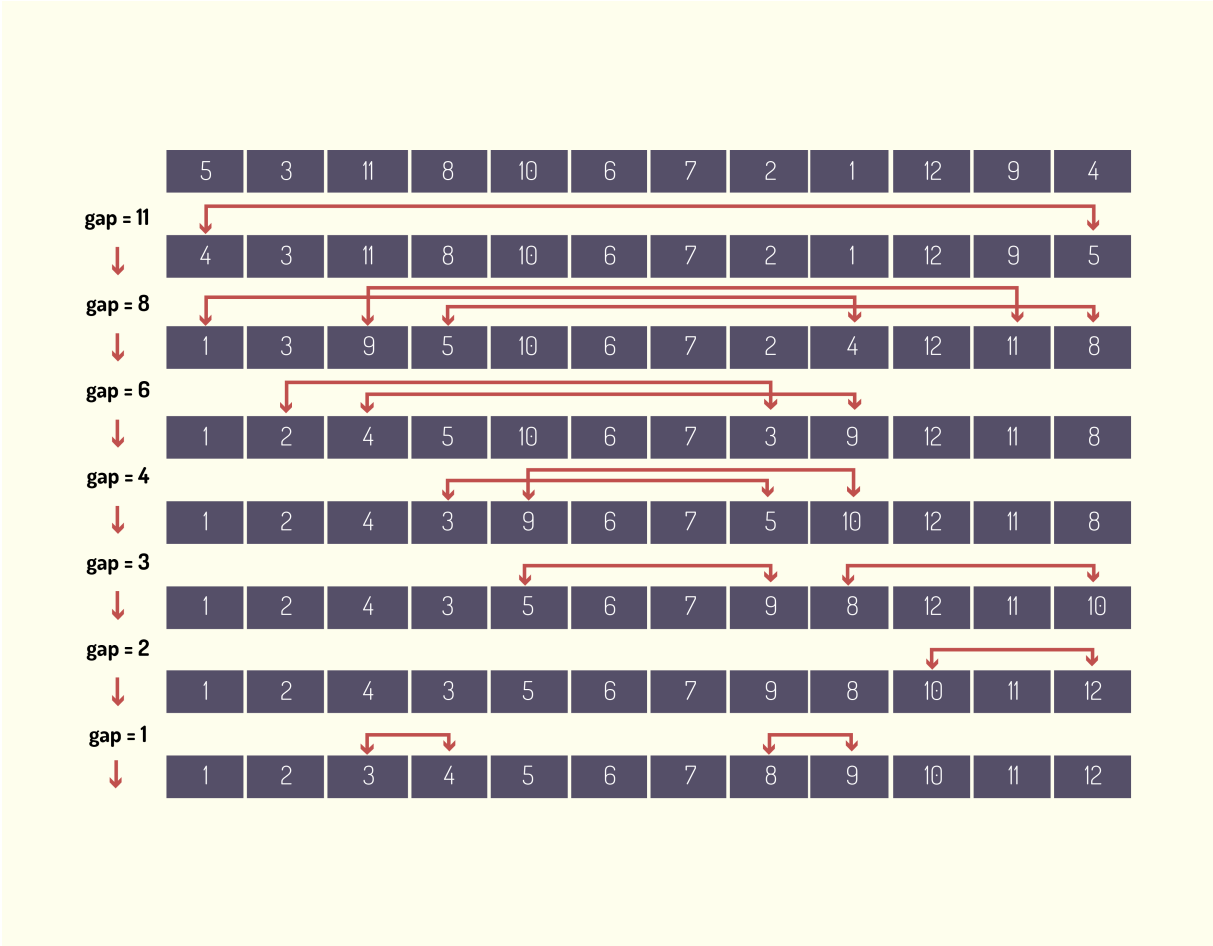
Eles são números muito baixos que se encontram no meio ou no final do vetor, e para ordenar esse elemento, se exigem diversas trocas em pares com o método BubbleSort. Para entendermos melhor, vamos utilizar como exemplo um vetor de 7 elementos:



3FIGURA 20.29 - Vetor de 7 posições com um turtle FONTE: o autor.

Veja, prezado(a) aluno(a), que o elemento 3 no vetor é quase o último, assim, necessita de várias trocas entre os elementos até que o número 3 se posicione no local correto. Agora que sabemos o que é um *turtle*, vamos entender como o algoritmo do Combsort funciona.

O Combsort se utiliza do conceito do GAP, igual ao QuickSort, o valor do GAP é obtido através de uma conta simples a cada volta do laço, GAP = GAP / 1.3, o retorno do valor sempre será um número inteiro, com isso, a cada volta, o algoritmo vai sempre diminuir até que fique 1. Ao se chegar nesse valor, é aplicado o conceito do BubbleSort, ou seja, se compara o elemento atual com o seu vizinho da direita. Esse GAP também é conhecido como fator de encolhimento.

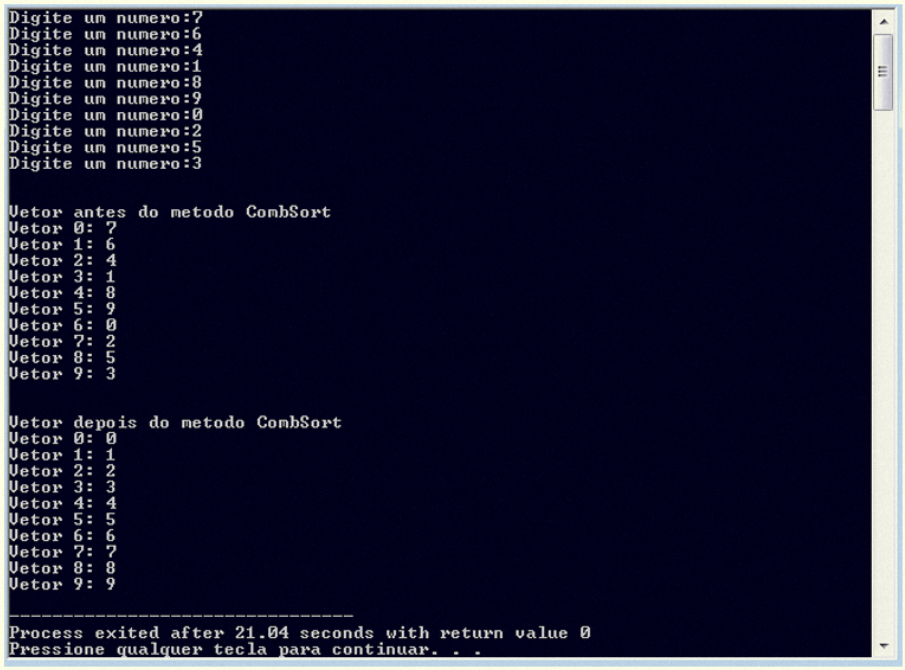


3FIGURA 21.29 - Método de funcionamento do CombSort FONTE: Comb Sort (2015, *on-line*).

Veja, caro(a) aluno(a), que o Combsort traz uma evolução significativa no BubbleSort aplicando essa remoção dos turtles, esse método, como os demais já apresentados, mescla dois conceitos para buscar uma melhoria na ordenação dos vetores. Observe agora um exemplo do CombSort em linguagem C.

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

int proximoGap(int gap, int n){

    gap = (gap*n)/13;
    if (gap < 1)
        return 1;
    return gap;
}

void combSort(int a[], int n){
    // Inicializando o gap
    int gap = n;

    // Inicializa a troca com 0
    int swapped = 0;

    // Se o gap for diferente de 1 ou a troca 0
    while (gap != 1 || swapped == 0){
        // proximo Gap
        gap = proximoGap(gap, n);
        swapped = 1;

        // Compração dos elementos
        for (int i = 0; i < n - gap; i++){
            int j = i + gap;
            if (a[i] > a[j]) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
                swapped  =  1;
            }
        }
    }
}
main(){
    int X[10], i;
    // inserindo os números no vetor
    for(i=0; i<=9; i++){
        printf("Digite um numero:");
        scanf("%d", &X[i]);
    }
    printf("\n\nVetor antes do metodo CombSort\n");
    for(i=0;i<=9;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
    // ordenando de forma crescente
    combSort(X, 10);

    printf("\n\nVetor depois do metodo CombSort\n");
    // monstrando o vetor ordenado
    for(i=0;i<=9;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
}
```

3FIGURA 22.29 - Script em linguagem C, CombSort FONTE: DEV C++.



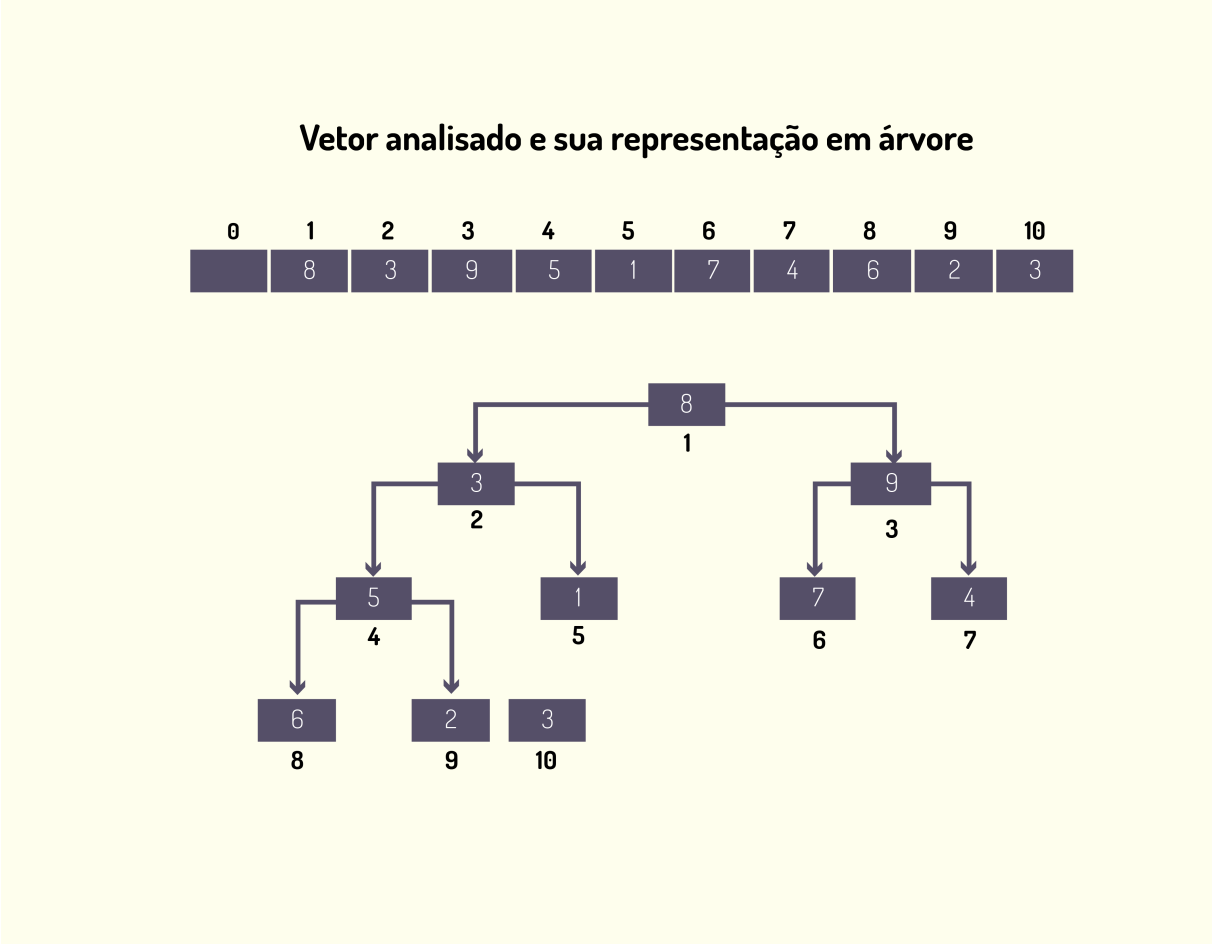
3FIGURA 23.29 - Resultado do programa CombSort FONTE: CombSort

HeapSort

A ordenação HeapSort também se utiliza do conceito de comparação entre elementos e troca, contudo a forma de pré-ordenação e ordenação dos elementos é uma das mais diferentes, pois se utiliza da estrutura de dados Heap. Segundo Ascencio e Araújo(2010 p.74), esse algoritmo nada mais é que:

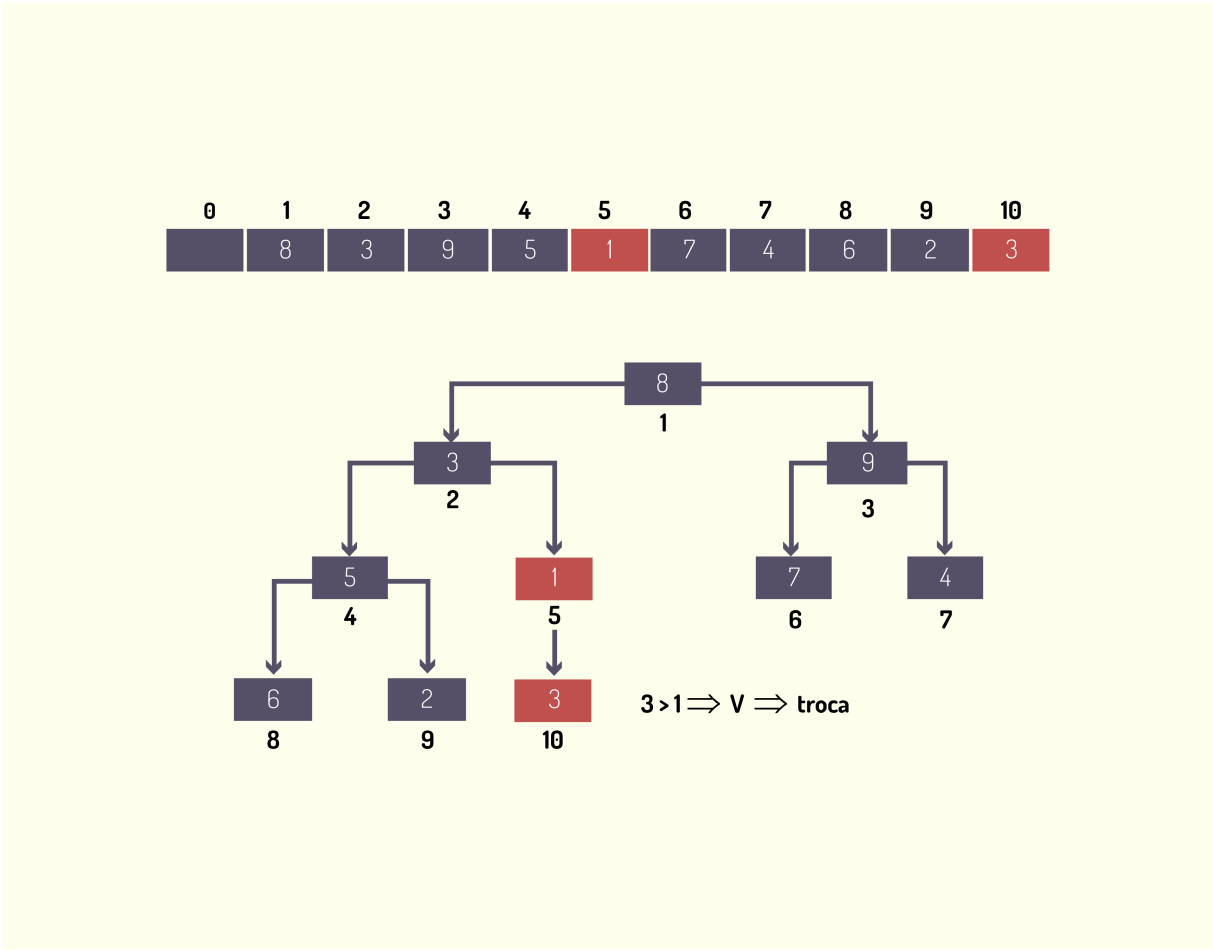
> um vetor (X) que pode ser visto como um árvore binária completa, onde cada nó possui no máximo 2 filhos. Cada vértice da árvore binária corresponde a um elemento do vetor. A árvore é completamente preenchida exceto no último nível. cada nível é sempre preenchido da direita para a esquerda.

Pode-se dizer que esse algoritmo se utiliza de duas etapas para efetuar a ordenação, pois, antes de executar as comparações, ele distribui os elementos do vetor em forma de árvore binária:

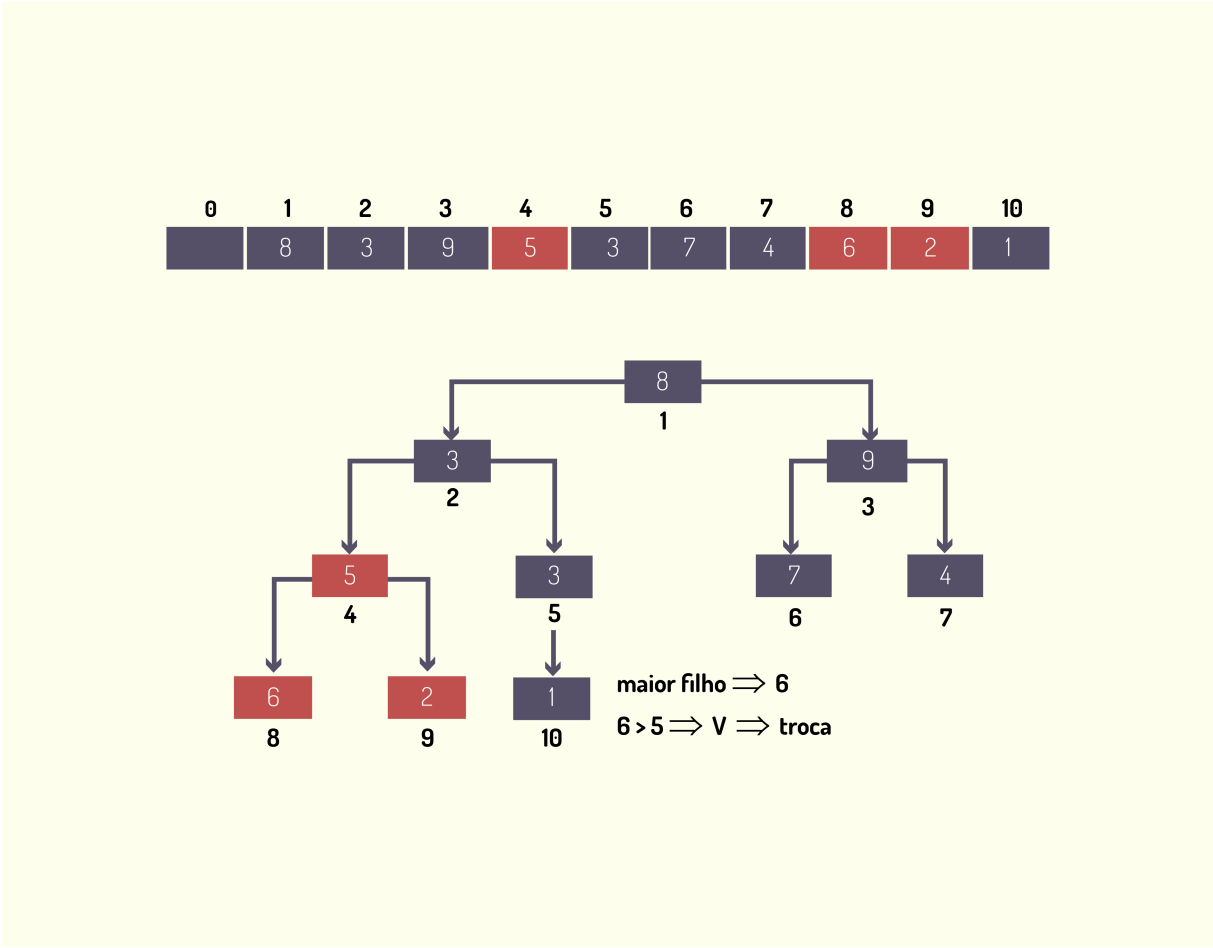


3FIGURA 24.29 - 1ª etapa: construção da árvore binária FONTE: Ascencio e Araújo (2010 p.75).

Após a árvore ser construída, se iniciam as comparações pelo nível que contém somente folhas, em que são comparados os elementos filhos com o pai, caso o valor do filho seja menor que o do pai, ambos são alterados na árvore, esse procedimento é efetuado em todos os níveis da árvore e com todos os filhos e pais.

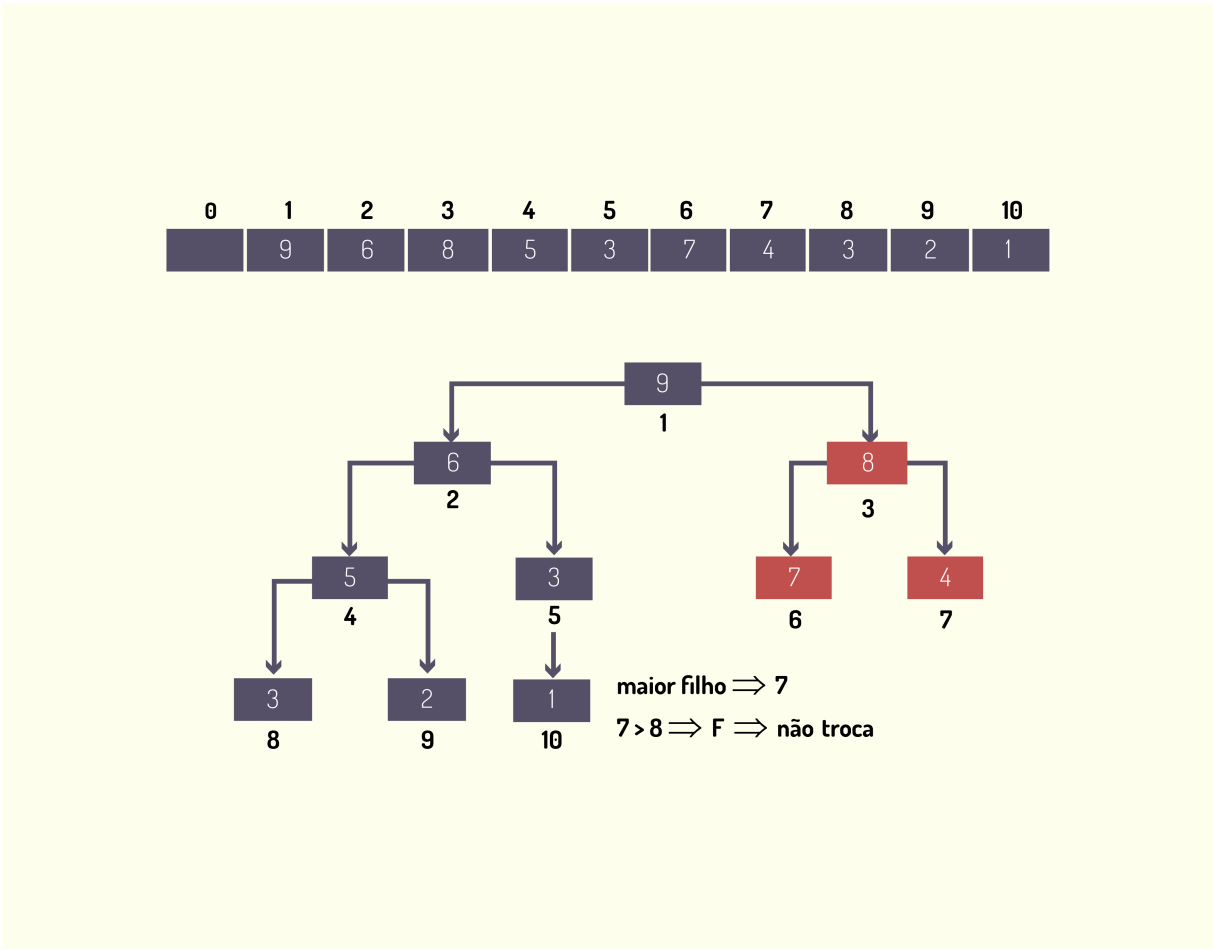


3FIGURA 25.29 - 1ª comparação entre as folhas e o nó pai FONTE: Ascencio e Araújo (2010 p.76).



3FIGURA 26.29 - 2ª comparação entre as folhas e o nó pai FONTE: Ascencio e Araújo (2010 p.76).

Ao terminar as comparações, os elementos com valores maiores vão estar mais ao topo da árvore, com a qual reordenada, se efetua uma outra troca entre o elemento raiz e o último elemento folha. Como o elemento raiz já era o mais alto quando foi feita a troca, ele já vai estar em sua posição correta e, neste momento, ele é eliminado da árvore e alocado no vetor.



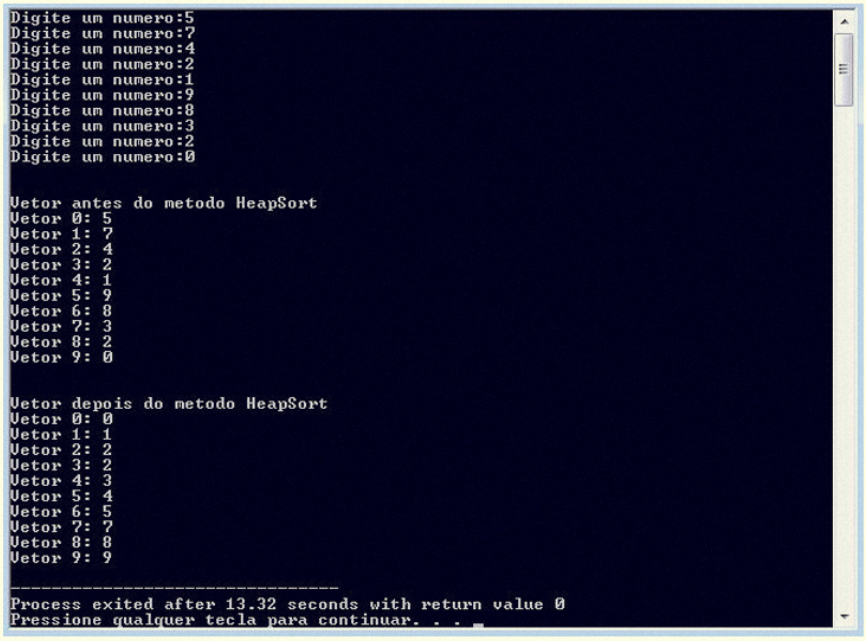
3FIGURA 27.29 - Com as trocas finalizadas, o último elemento é alterado com o primeiro elemento FONTE: Ascencio e Araújo (2010 p.77).

Após esse procedimento, a raiz se mantém e a folha é removida, assim reiniciando todo o processo até que toda a árvore seja realocada no vetor em ordem crescente.

Caro(a) aluno(a), vejamos a seguir um exemplo do método Heapsort, utilizando a linguagem C.

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

void heapsort(int a[], int n) {
   int i = n / 2, pai, filho, t;
   for (;;) {
      if (i > 0) {
         i--;
         t = a[i];
      } else {
         n--;
         if (n == 0) return;
         t = a[n];
         a[n] = a[0];
      }
      pai = i;
      filho = i * 2 + 1;
      while (filho < n) {
         if ((filho + 1 < n)  &&  (a[filho + 1] > a[filho]))
            filho++;
         if (a[filho] > t) {
            a[pai] = a[filho];
            pai = filho;
            filho = pai * 2 + 1;
         } else {
            break;
         }
      }
      a[pai] = t;
   }
}

main(){
    int X[10], i;
    // inserindo os números no vetor
    for(i=0; i<=9; i++){
        printf("Digite um numero:");
        scanf("%d", &X[i]);
    }
    printf("\n\nVetor antes do metodo HeapSort\n");
    for(i=0;i<=9;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
    // ordenando de forma crescente
    heapsort(X, 10);

    printf("\n\nVetor depois do metodo HeapSort\n");
    // monstrando o vetor ordenado
    for(i=0;i<=9;i++){
        printf("Vetor %d: %d \n", i, X[i]);
    }
}
```

3FIGURA 28.29 - Script em linguagem C, HeapSort FONTE: DEV C++.



3FIGURA 29.29 - Resultado do programa em C, HeapSort FONTE: HeapSort.

Indicação de leitura

Nome do livro: Estruturas de dados

Editora: Pearson education do Brasil

Autor: Ascencio, Ana Fernanda e Araújo, Graziela Santos de Gomes

ISBN: 8576058812

ISBN: 13: 9788576058816

Esse livro tem dois objetivos claros: ser um forte aliado dos professores da área de computação, ajudando-os a levar os estudantes a analisar diversas soluções para um mesmo problema, e contribuir para o desenvolvimento de profissionais que querem adotar padrões elevados de qualidade no desempenho de suas atividades. Para atingir esses objetivos, ele aborda estruturas de dados e análise da complexidade, bem como análise de algoritmos e sua representação gráfica (os grafos), de maneira

bastante didática, altamente explicativa. Além disso, traz as implementações e os códigos tanto em Java como em C e C++, permitindo ao leitor entender não apenas a aplicação prática dessas linguagens, mas também suas semelhanças e diferenças.

UNIDADE IV

Técnicas de Busca de Dados

Carlos Danilo Luz

Caro(a) aluno(a)! Seja bem-vindo(a)! Nesta última unidade, vamos aprender métodos simples mais muito eficazes de busca e classificação de dados. Será apresentada a você a forma de encontrar elementos em vetores organizados ou não, sendo aplicada a indexação. Estudaremos o conceito de busca binária em vetores ordenados, além de analisarmos como é efetuada a busca de elementos em estrutura de árvore binária. Um dos métodos a serem apresentados é a Tabela Hasc, que tem como função o agrupamento de dados para efetuar busca. Ao final de cada tópico, será vista uma implementação dos conceitos em linguagem C para melhor entendimento.

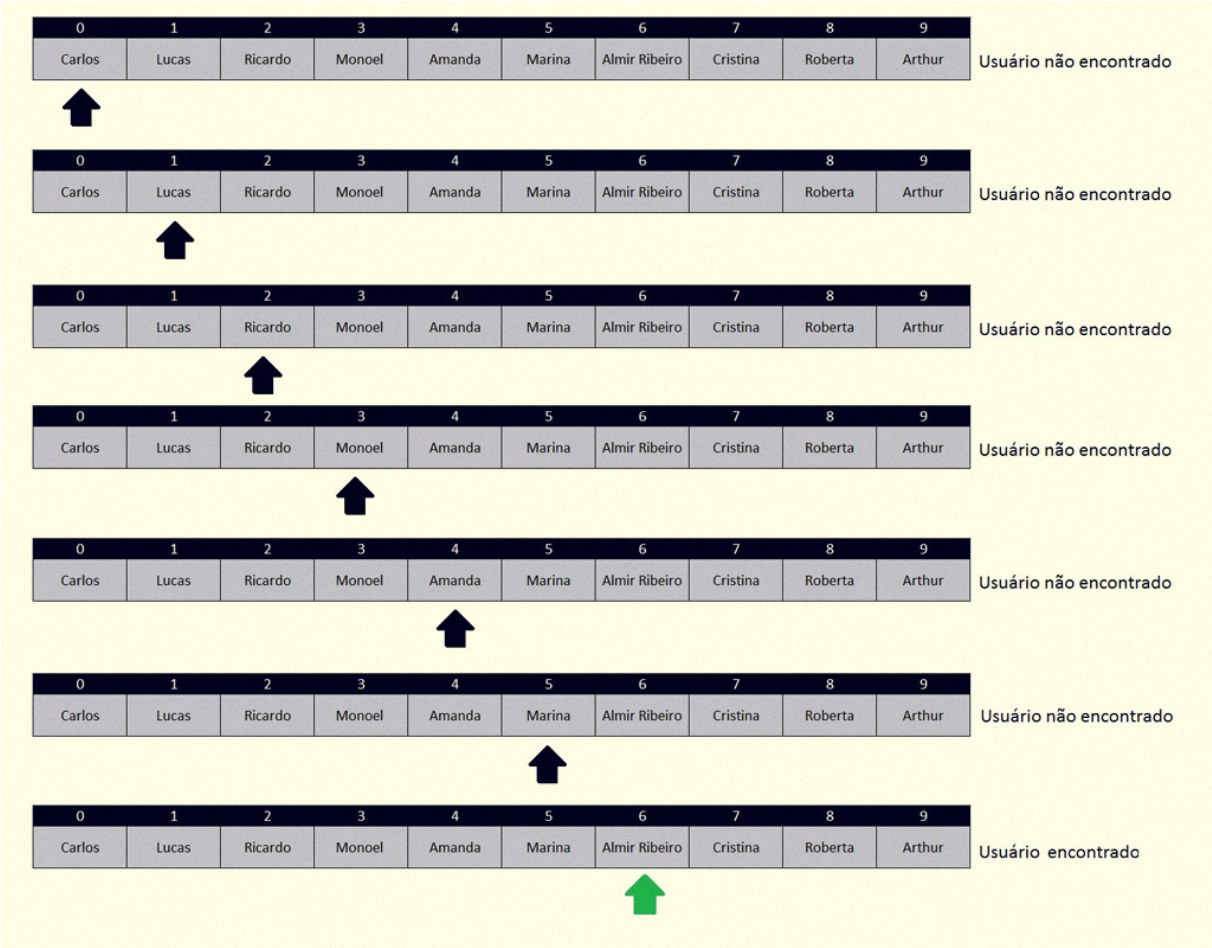
Buscas Sequenciais

Vimos nas unidades anteriores diversas formas de busca e ordenação de dados, algumas técnicas mais complexas e outras mais simples. Podemos dizer que a busca sequencial é uma das formas mais simples e, ao mesmo tempo, uma das mais eficazes quando precisamos buscar algum elemento em um vetor.

Busca Sequencial

A busca sequencial se baseia em varrer um vetor por inteiro de seu início ao fim, não sendo considerados os ponteiros, como visto nas listas encadeadas e duplamente encadeadas, em que esses informam qual o próximo elemento da lista mesmo se os elemento não estão ordenados. Esse método é bem simples, sendo um dos mais utilizados ao efetuar busca de elementos em vetores ou lista lineares, podendo esses estar de forma ordenada ou não ordenada.

Imagine que temos um vetor com 10 posições de forma não ordenada, em que, em cada posição, temos o nome de um usuário de nosso sistema, precisamos encontrar o registro do usuário **Almir Ribeiro,** caso ele exista no vetor, temos que retornar à posição em que ele se encontra, se o usuário estiver na posição a ser analisada, a busca percorre até encontrar. Vejamos a representação do algoritmo na Figura 4.1.

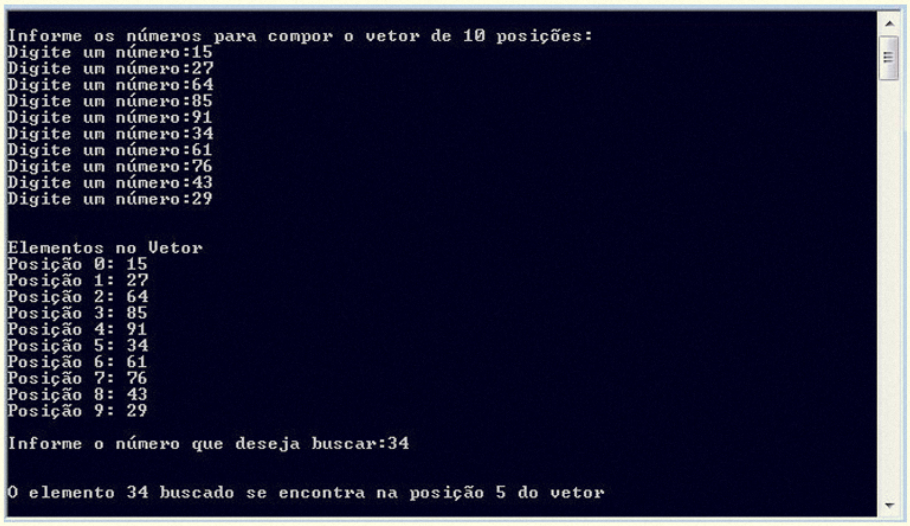


4FIGURA 1.29 - Representação da busca sequencial FONTE: o autor

Veja, prezado(a) aluno(a), que a cada nova iteração é efetuada uma comparação com o próximo elemento do vetor, a quantidade de iterações vai depender do tamanho do vetor e da posição do elemento no vetor. Agora que você já entendeu como funciona a busca, vamos observar um exemplo em linguagem C utilizando um vetor com números.

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

int vetor[10], i, itemBusca, posicao;
// inserindo os números no vetor

main(){
    setlocale(LC_ALL, "Portuguese"); //habilita a acentuação
    printf("\nInforme os números para compor o vetor de 10 posições:\n");
    for(i=0; i<=9; i++){
        printf("Digite um número:");
        scanf("%d", &vetor[i]);
    }

    printf("\n\nElementos no Vetor\n");
    for(i=0;i<=9;i++){
        printf("Posição %d: %d\n", i, vetor[i]);
    }

    printf("\nInforme o número que deseja buscar:");
    scanf("%d",&itemBusca);

    for(i=0;i<=9;i++){
        if(vetor[i]==itemBusca){
            posicao=i;
        }
    }

    if(!posicao){
        printf("\n\nO elemento %d buscado não existe no vetor\n\n", itemBusca);
    }else{
        printf("\n\nO elemento %d buscado se encontra na posição %d do vetor\n\n", itemBusca, posicao);
    }
}
```

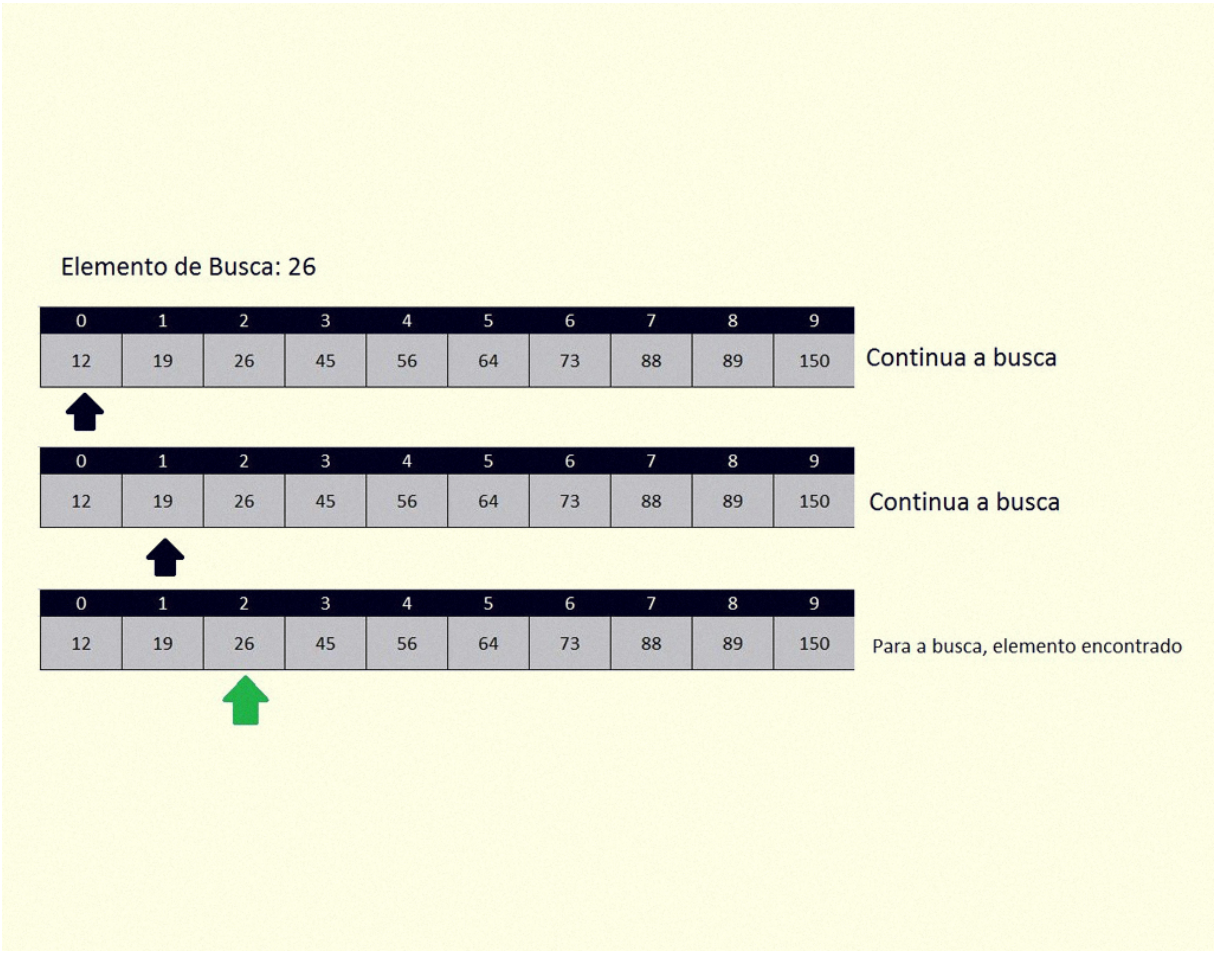
4FIGURA 2.29 - Script em linguagem C, busca sequencial FONTE: DEV C++.



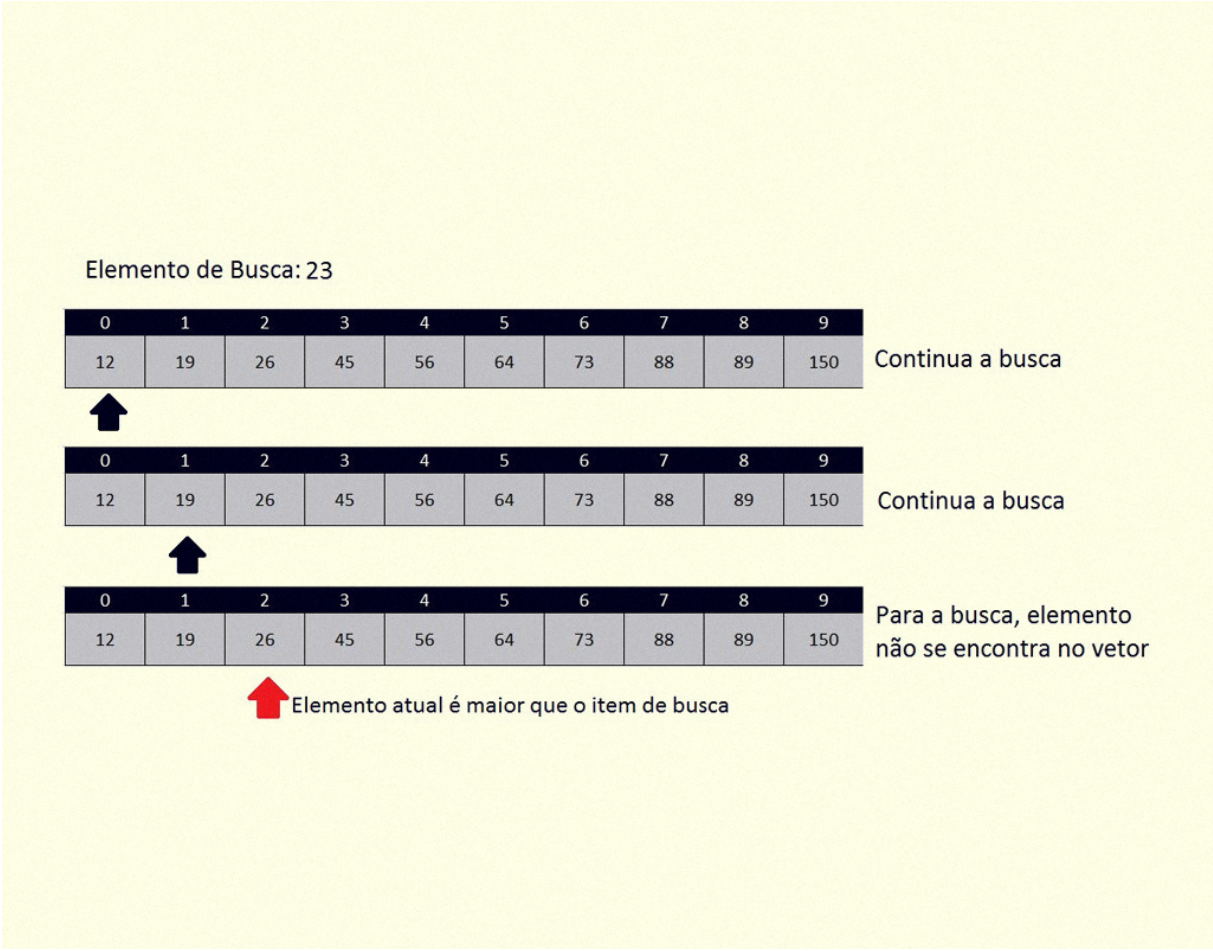
4FIGURA 3.29 - Resultado do programa de busca sequencial FONTE: DEV C++

Pode-se compreender que a execução desse tipo de algoritmo é bem simples e muito eficaz, mas podemos nos deparar com vetor muito extenso de forma que o algoritmo tem que efetuar a varredura total do vetor antes de informar se o elemento que estamos buscando está presente ou não. Uma solução muito simples que resolve esse problema é, antes de efetuar a busca, ordenar o vetor, podendo ser utilizado qualquer um dos métodos estudados na unidade anterior.

Ao efetuar esse processo de ordenação, temos um ganho de desempenho computacional, já que o vetor não será varrido por completo. Assim que encontrarmos o elemento buscado, retorna-se à posição do elemento que estava sendo buscado, caso o valor buscado seja menor que o elemento atual, a busca é interrompida e será informado que o elemento não existe no vetor.



4FIGURA 4.29 - Busca sequencial com vetor ordenado, elemento encontrado FONTE: o autor.

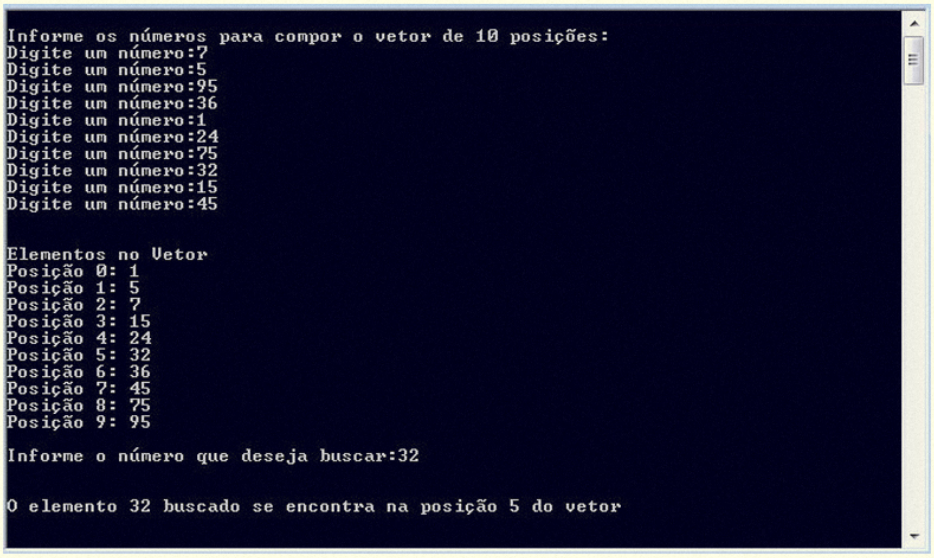


4FIGURA 5.29 - Busca sequencial com vetor ordenado, elemento não encontrado FONTE: o autor.

No exemplo apresentado, a busca se encerrou com três iterações, pois, no primeiro caso, o elemento 26 se encontrava na 3 posição; o segundo caso também se encerrou na terceira iteração, pois o elemento atual era maior do que o elemento de busca, consequentemente, este não estará daquela posição em diante, já que o vetor está ordenado. Vejamos um exemplo em linguagem C, para ordenar o vetor, será utilizado o método ShellSort.

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

int vetor[10], i, j, value, gap = 1, itemBusca, posicao;
// inserindo os números no vetor

main(){
    setlocale(LC_ALL, "Portuguese"); //habilita a acentuação
    printf("\nInforme os números para compor o vetor de 10 posições:\n");
    for(i=0; i<=9; i++){
        printf("Digite um número:");
        scanf("%d", &vetor[i]);
    }
        // ordenando de forma crescente ShellSort
        while(gap < 8) {
            gap = 3*gap+1;
        }
        while ( gap > 1) {
            gap /= 3;
            for(i = gap; i <=9; i++) {
                value = vetor[i];
                j = i - gap;
                while (j >= 0 && value < vetor[j]) {
                    vetor[j + gap] = vetor[j];
                    j -= gap;
                }
                vetor[j + gap] = value;
            }
        }

    printf("\n\nElementos do Vetor ordenados \n");
    for(i=0;i<=9;i++){
        printf("Posição %d: %d\n", i, vetor[i]);
    }


    printf("\nInforme o número que deseja buscar:");
    scanf("%d",&itemBusca);

    for(i=0;i<=9;i++){
        if(vetor[i]==itemBusca){
            posicao=i;
            printf("\n\nO elemento %d buscado se encontra na posição %d do vetor\n\n", itemBusca, posicao);
            break;//força a saída imediata do loop parando o código

        }else if(itemBusca<vetor[i]){
            printf("\n\nO elemento %d buscado não existe no vetor\n\n", itemBusca);
            break;//força a saída imediata do loop parando o código
        }
    }

}
```

4FIGURA 6.29 - Script em linguagem C, busca sequencial com vetor ordenado FONTE: DEV C++.



4FIGURA 7.29 - Resultado do programa de busca sequencial com o vetor ordenado FONTE: DEV C++

Ao compararmos as duas formas de busca, fica bem claro que ordenar o vetor antes de efetuar a busca torna o script mais eficaz e rápido para trazer o retorno do elemento. Outra forma interessante de efetuarmos as buscas é através de indexes, item que veremos a seguir.

¶ Reflita

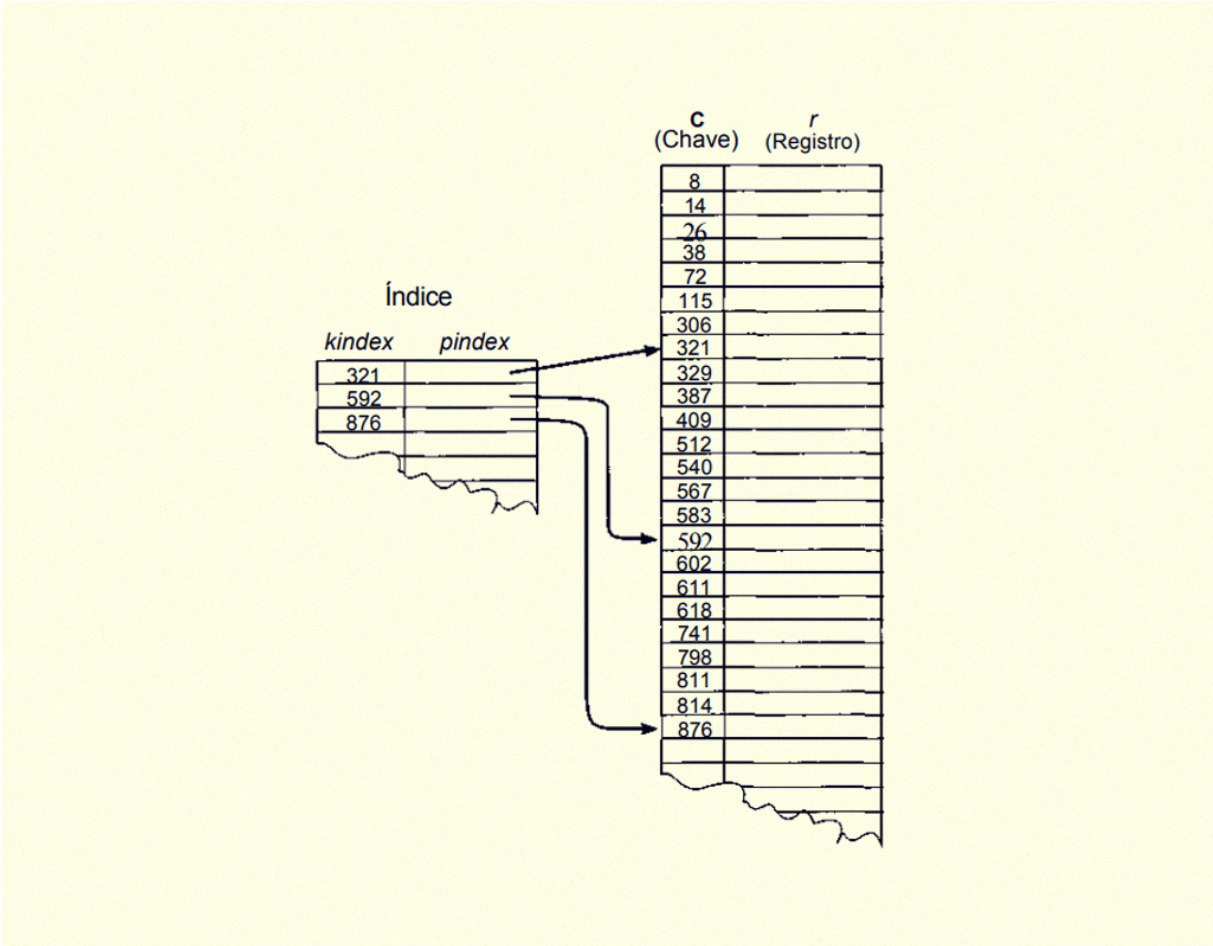
Conforme visto, ao se utilizar um vetor ordenado antes de efetuar a busca sequencial, o desempenho computacional é melhor, pois se utilizam menos iterações. No exemplo mostrado, foi utilizado um vetor contendo números, se fosse um vetor com nomes de usuários, seria possível efetuar essa busca com o vetor ordenado?

Busca Sequencial Indexada

O método que se aplica para a busca sequencial indexada é bem parecido com a busca apresentada anteriormente, mas é utilizada uma tabela auxiliar, chamada de "tabela índices". No algoritmo sequencial simples, era possível efetuar a busca em um vetor ou lista com os elementos não ordenados, na busca sequencial indexada, só é possível utilizar esse tipo de busca em lista ou vetores totalmente ordenados. Tenenbaum (1995 p.498) complementa dizendo que:

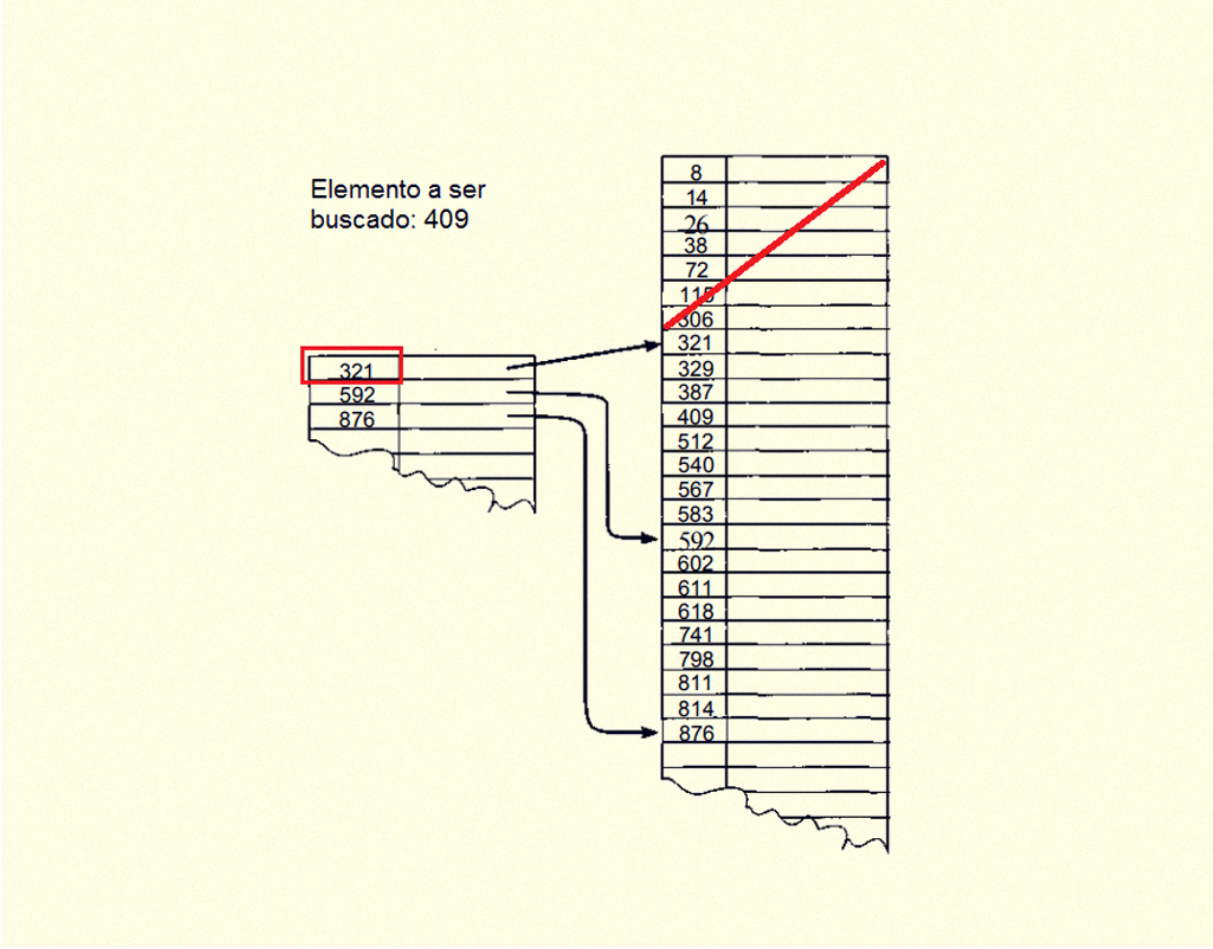
> Cada elemento no índice consiste em uma chave kindex e um ponteiro para o registro no arquivo que corresponde a kindex. Os elementos no índice e no arquivo devem ser classificados pela chave. Se o índice for um oitavo do tamanho do arquivo, todo oitavo registro do arquivo será representado no índice.

O intervalo entre os elementos a serem classificados como índices pode variar de acordo com o tamanho do vetor ou pela forma como foi desenvolvido o programa. Para que possamos entender melhor como é montada essa tabela e também como é efetuada a separação dos elementos que serão índices, vamos observar a Figura 4.8:



4FIGURA 8.29 - Composição da tabela de índices FONTE: Tenenbaum (1995 p.499).

Na busca apenas sequencial, essa se inicia diretamente no vetor que será varrido, iniciando pela posição zero, após isso, o algoritmo de busca irá percorrer o vetor. Na busca indexada, antes de se iniciar a busca no vetor, a tabela de índices é consultada e é verificado em qual o ponto a busca deve ser iniciada, sendo assim, essa não precisa ser feita desde o começo, tornando a busca mais rápida e otimizada.



4FIGURA 9.29 - Consulta na tabela de tabela de índices FONTE: adaptada de Tenenbaum (1995 p.499).

Ao consultar a tabela de índices, é observado que o elemento a ser buscado (409) é maior que o primeiro índice, a busca se inicia a partir desse ponto descartando todos os elementos anteriores, se o elemento for encontrado, a busca se encerra, caso o atual elemento seja maior que o buscado, também se encerra. Para compreender melhor seu algoritmo, vejamos um exemplo se trata de uma implementação da busca sequencial indexada em linguagem C.

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

int vetor[15], i, j, select, menor, troc, itemBusca, posicao;
int tabelaindices[15], posicaovetor[15];
// inserindo os números no vetor

main(){
	setlocale(LC_ALL, "Portuguese"); //habilita a acentuação
	printf("\nInforme os números para compor o vetor de 15 posições:\n");
	for(i=0; i<=14; i++){
		printf("Digite um número:");
		scanf("%d", &vetor[i]);
	}

	// ordenando de forma crescente com SelectionSort
	for(i=0;i<=13;i++){
		select = vetor[i];
		menor = vetor[i+1];
		troc = i+1;

			for(j=i+1;j<=14;j++){
		if(vetor[j] < menor) {
			menor = vetor[j];
			troc=j;
		}
	}
		if(menor < select){
			vetor[i]=vetor[troc];
			vetor[troc] = select;
		}
	}

	printf("\n\nElementos ordenados no Vetor \n");
	for(i=0;i<=14;i++){
		printf("Posição %d: %d\n", i, vetor[i]);
	}

	int intervaloIndices = 15/3; // total de registro dividido pela quantidade de indices desejado

	printf("\n\nTabela de indices:\n");
	for(i=0; i<=2; i++){
		tabelaindices[i]=vetor[intervaloIndices*i];
		posicaovetor[i]=(intervaloIndices*i);
		printf("Posição %d: %d, %d\n", i, tabelaindices[i], posicaovetor[i]);
	}

	printf("\nInforme o número que deseja buscar:");
	scanf("%d",&itemBusca);

	for(i=0; i<=2; i++){

		if(itemBusca>tabelaindices[i]){

		for(j=0; j<=intervaloIndices; j++){

		if(vetor[posicaovetor[i]+j]==itemBusca){
			posicao=posicaovetor[i]+j;
			printf("\n\nO elemento %d buscado se encontra na posição %d do vetor\n\n", itemBusca, posicao);
			break;//força a saída imediata do loop parando o código

		}else if(itemBusca<vetor[posicaovetor[i]+j]){
			printf("\n\nO elemento %d buscado não existe no vetor\n\n", itemBusca);
			break;//força a saída imediata do loop parando o código
		}
		}

	}

	}

}
```

4FIGURA 10.29 - Resultado da busca indexada FONTE: DEV C++

Perceba, prezado(a) aluno(a), que, ao utilizar esse método, temos um ótimo resultado se o compararmos com a busca simples, pois é possível encontrar o elemento buscado com menos iteração e com maior rapidez.

Buscas Binárias

A busca sequencial e indexada já trazem um retorno da busca de forma muito interessante, mas podemos melhorar ainda mais a forma de efetuar busca em vetores com menos esforço computacional, métodos esses que serão apresentados a seguir.

Busca binária

Podemos dizer que a busca binária é a forma mais eficiente de efetuar uma busca em um vetor ou arquivo ordenado, esse método não se utiliza de tabelas auxiliares como a busca indexada.

O método consiste em iniciar a busca pelo meio, caso o elemento buscado seja encontrado, a busca se encerra com apenas uma iteração, se o item de busca não estiver no meio do vetor, este é dividido em duas partes: à esquerda, ficam os elemento menores; à direita, os maiores. Caso o item de busca for menor que o meio, toda a parte da direita é descartada, se iniciando o processo novamente com o menor vetor.

A cada iteração, a busca binária elimina a quantidade de itens a serem varridos para busca, vale ressaltar que esse método só funciona com vetor ordenados.

Para compreendermos melhor, vamos simular a busca em um vetor com 10 posições, o elemento a ser buscado é o 7, para descobrirmos qual é o meio do vetor, vamos efetuar um cálculo simples, sabendo que um vetor se inicia com a posição 0 e termina com o tamanho do vetor-1 no caso 10-1 = 9, a última posição será 9.

$meio~=~\frac{\left(menor~+~maior \right)}{2}$

$meio~=~\frac{\left(0~+~9 \right)}{2}$

$meio~=~\frac{\left(~9 \right)}{2}$

$meio~=~4,5$

A variável meio retornou o valor de 4,5, mas as posições de um vetor são números inteiros, por isso, o resultado é convertido para um número inteiro, sendo retornado o número 4, dessa forma, a busca se iniciou pela quarta posição do vetor.

Se o número 7 não for encontrado e supondo que o elemento na quarta posição seja 19, o processo se inicia na parte menor, efetuando o cálculo do meio novamente, mas agora com quatro posições.

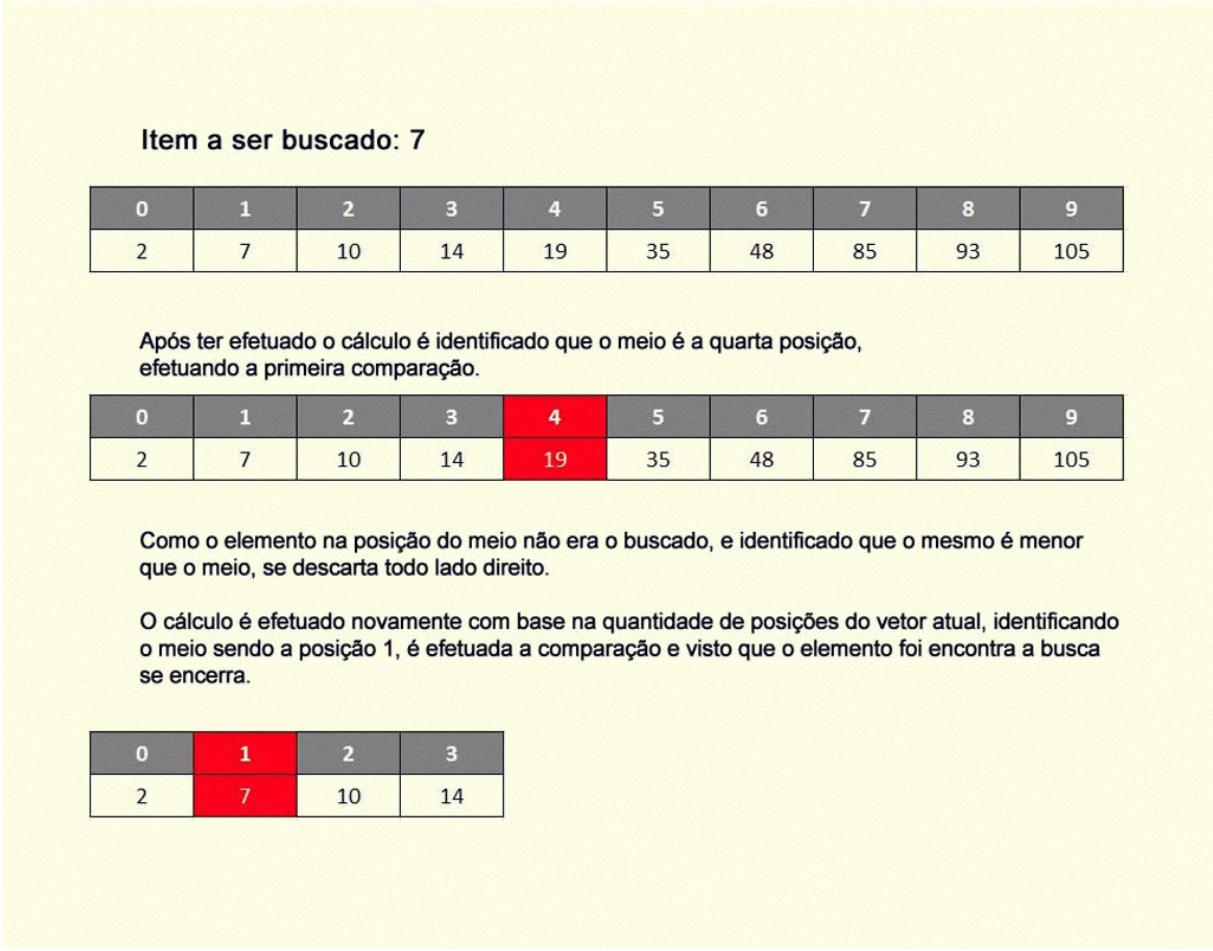
$meio~=~\frac{\left(menor~+~maior \right)}{2}$

$meio~=~\frac{\left(0~+~3 \right)}{2}$

$meio~=~\frac{\left(3~ \right)}{2}$

$meio~=~1,5$

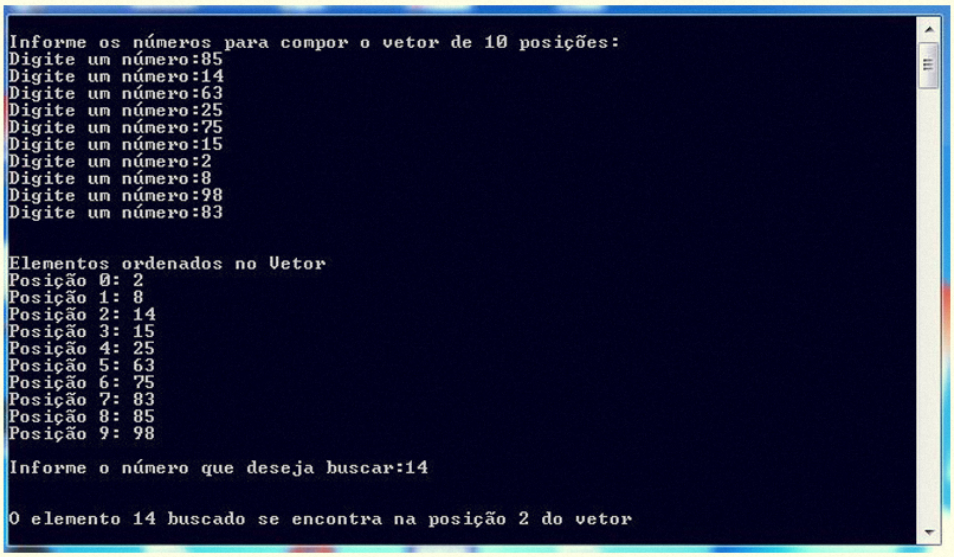
Novamente, o retorno da variável meio é convertido para um número inteiro, sendo o meio a posição um do vetor, caso o elemento encontrado seja este, a busca se encerra, vejamos a demonstração dessa busca na Figura 4.12.



4FIGURA 11.29 - representação da busca binário em um vetor com 10 posições FONTE: autor.

Perceba, prezado(a) aluno(a), que foram necessárias apenas 2 iterações para encontrar o item a ser buscado, sendo assim, ganhou-se performance e agilidade na busca. Observamos também o exemplo da busca binária em linguagem C.

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

int vetor[10], i, j, select, menor, troc, itemBusca;
// inserindo os números no vetor

int BuscaBinaria (int vet[], int itemBusca, int TamVetor)
{
        int inf = 0;    // Tamanho inicial do vetor
        int sup = TamVetor-1; // Tamanho total do vetor
        int meio;
        while (inf <= sup)
        {
        meio = (inf + sup)/2;
        if (itemBusca == vet[meio]){
                 return meio;
                 break;//força a saída imediata do loop parando o código
}

        if(itemBusca < vet[meio]){
                sup = meio-1;
        } else{
                inf = meio+1;
        }
        }
        return -1;
}

main(){
setlocale(LC_ALL, “Portuguese”); //habilita a acentuação
printf(“\nInforme os números para compor o vetor de 10 posições:\n”);
for(i=0; i<=9; i++){
         printf(“Digite um número:”);
         scanf(“%d”, &vetor[i]);
}


// ordenando de forma crescente com SelectionSort
for(i=0;i<=8;i++){
         select = vetor[i];
         menor = vetor[i+1];
         troc = i+1;

         for(j=i+1;j<=9;j++){
                 if(vetor[j] < menor) {
                         menor = vetor[j];
                         troc=j;
                 }
         }
         if(menor < select){
                 vetor[i]=vetor[troc];
                 vetor[troc] = select;
         }
}

printf(“\n\nElementos ordenados no Vetor \n”);
for(i=0;i<=9;i++){
         printf(“Posição %d: %d\n”, i, vetor[i]);
}

printf(“\nInforme o número que deseja buscar:”);
scanf(“%d”,&itemBusca);

int retorno = BuscaBinaria(vetor,itemBusca,10);
if(retorno==-1){
         printf(“\n\nO elemento %d buscado não existe no vetor\n\n”, itemBusca);
}else{
         printf(“\n\nO elemento %d buscado se encontra na posição %d do vetor\n\n”, itemBusca, retorno);
}

}
```



4FIGURA 12.29 - Resultado do exemplo de busca binária em C FONTE: DEV C++

Busca por Interpolação

A busca por interpolação é também um extensão da busca sequencial, sendo aplicada sobre vetores ordenados. O método dessa busca é muito semelhante à busca binária que acabamos de estudar, mas ela é aplicada se os valores no vetor estiverem distribuídos de forma uniforme entre a posição inicial e final do vetor, podendo ser mais eficiente que a binária.

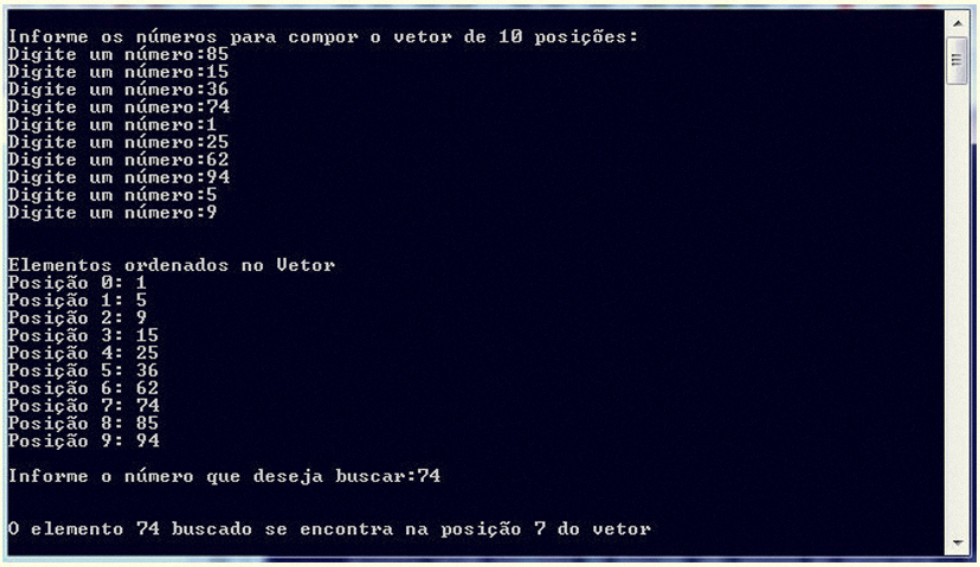
Sabendo que os valores estão distribuídos de maneira uniforme, vamos utilizar **o item de busca (arg)**, para que possamos identificar uma posição aproximada do elemento que buscamos. Assim como na busca binária, iniciamos efetuando um cálculo para encontrar o meio inicial.

$meio~=~menor~+~\left(maior~-~menor \right)*(\frac{arg~-~vec\left[menor \right]}{~vec\left[maior \right]~-~vec\left[menor \right]~}$)

A diferença entre as buscas é que, através do cálculo apresentado, a busca por interpolação já se inicia na parte superior ou inferior do vetor. Se arg for menor que o meio, a informação do maior será atualizado **meio -1**, assim, a busca acontecerá somente na parte inferior do vetor, caso contrário, a variável será **meio +1**, sendo efetuada a busca na parte superior. O código a seguir demonstra a implementação da busca por interpolação em linguagem C.

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

int vetor[10], i, j, select, menor, troc, itemBusca;
// inserindo os números no vetor

int buscaInter(int vect[], int itemBusca, int tam){
int menor, maior, meio = 0, busc;
menor = 0;
maior = tam-1;
busc = -1;

while((menor <= maior) && (busc == -1)){
meio = menor + (maior - menor) * ((itemBusca - vect[menor])/(vect[maior] - vect[menor]));

if (itemBusca == vect[meio]){
         busc = meio;
}
if(itemBusca < vect[meio]){
         maior = meio - 1;
}else{
         menor = meio + 1;
}
}
return(busc);
}

main(){
setlocale(LC_ALL, “Portuguese”); //habilita a acentuação
printf(“\nInforme os números para compor o vetor de 10 posições:\n”);
for(i=0; i<=9; i++){
         printf(“Digite um número:”);
         scanf(“%d”, &vetor[i]);
}


// ordenando de forma crescente com SelectionSort
for(i=0;i<=8;i++){
         select = vetor[i];
         menor = vetor[i+1];
         troc = i+1;

         for(j=i+1;j<=9;j++){
                  if(vetor[j] < menor) {
                           menor = vetor[j];
                           troc=j;
                  }
         }
         if(menor < select){
                  vetor[i]=vetor[troc];
                  vetor[troc] = select;
         }
}

printf(“\n\nElementos ordenados no Vetor \n”);
for(i=0;i<=9;i++){
         printf(“Posição %d: %d\n”, i, vetor[i]);
}

printf(“\nInforme o número que deseja buscar:”);
scanf(“%d”,&itemBusca);

int retorno = buscaInter(vetor,itemBusca,10);
if(retorno==-1){
         printf(“\n\nO elemento %d buscado não existe no vetor\n\n”, itemBusca);
}else{
         printf(“\n\nO elemento %d buscado se encontra na posição %d do vetor\n\n”, itemBusca, retorno);
}
```

```
}
```



4FIGURA 13.29 - Resultado do programa de busca por interpolação FONTE: DEV C++

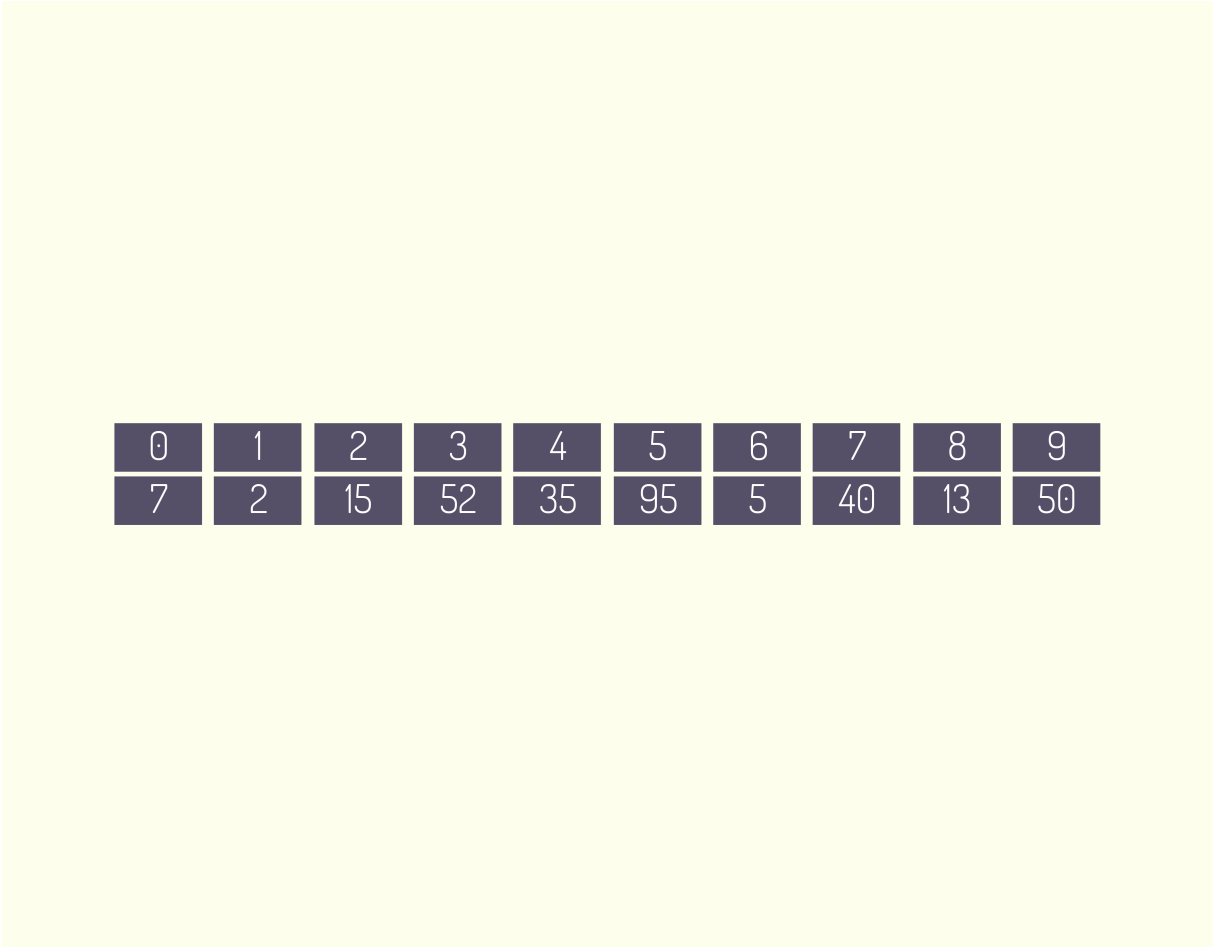
Buscas em árvores binárias e Montagem de árvores binárias

Outro método de busca a ser utilizado é a busca por meio de árvores binárias, em que buscamos um elemento dentro de uma árvore, sendo também possível montar uma árvore binária para a busca através de elementos de um vetor, ordenado ou não ordenado. Vejamos agora como isso é efetuado.

Montando Árvores Binárias

Vimos na unidade II um tipo de estrutura muito interessante que é utilizada para organizar os dados, sendo considerada uma excelente estrutura pelo fácil acesso às informações, estamos falando da árvore binária. Vimos que há várias estruturas, tais como a árvore binária simples, a completa, a estritamente binária, entre outras.

Agora que você relembrou o que são as árvores binárias, vamos aprender a montar uma árvore binária a partir de um vetor, estando este ordenado ou não. Vamos utilizar como exemplo um vetor de 10 posições não ordenado.



4FIGURA 14.29 - vetor com 10 posições FONTE: o autor.

Como já visto, toda árvore tem a sua raiz, então, nosso primeiro passo é identificar no vetor o elemento raiz. Para encontrar a posição do elemento raiz, vamos utilizar o mesmo cálculo efetuado para encontrar o meio do vetor na busca binária, sendo que vetor se inicia com a posição 0 e termina com o tamanho do **vetor-1,** no caso ,10 -1 = 9, assim, a última posição será 9.

$meio~=~\frac{\left(menor~+~maior \right)}{2}$

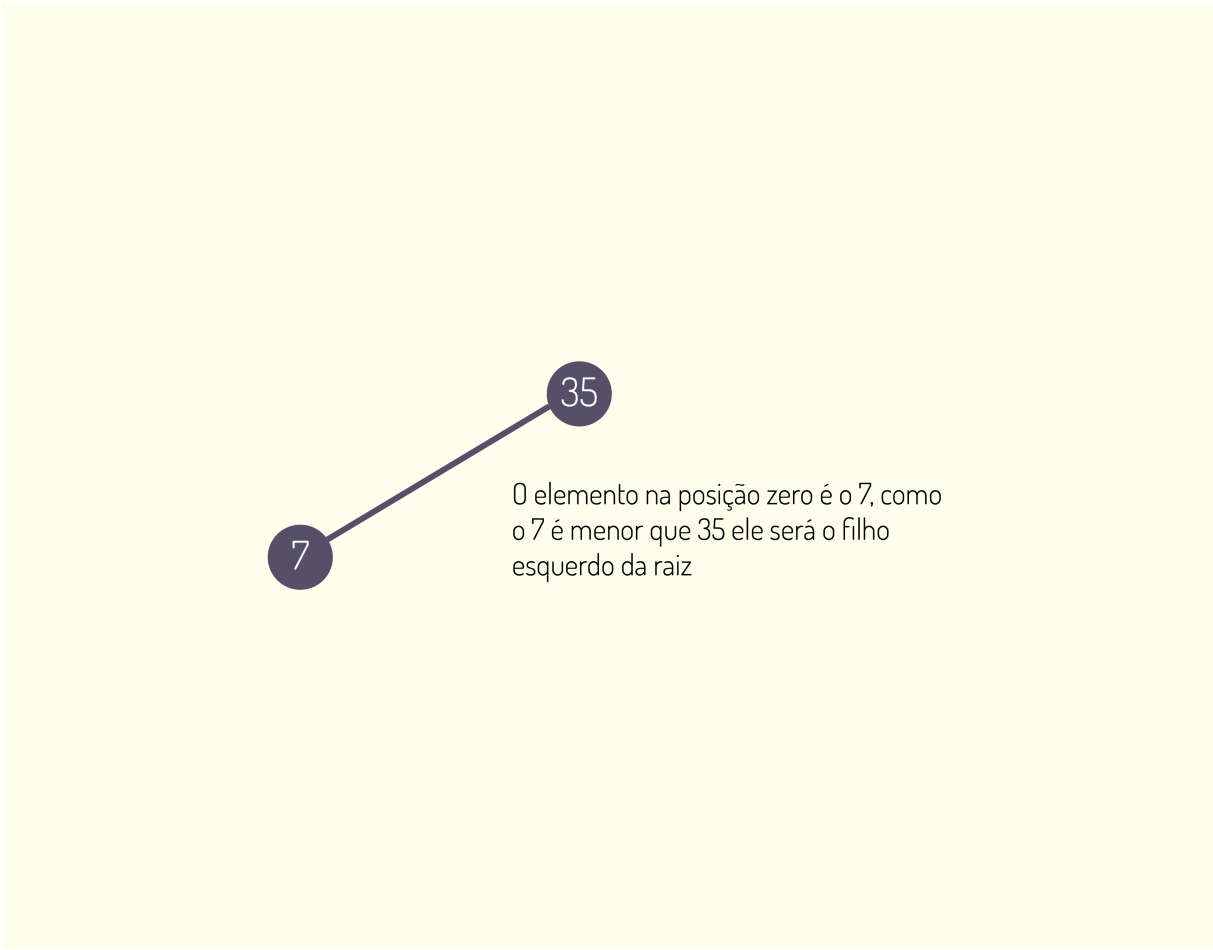
$meio~=~\frac{\left(0~+~9 \right)}{2}$

$meio~=~\frac{\left(9~ \right)}{2}$

$meio~=~4,5$

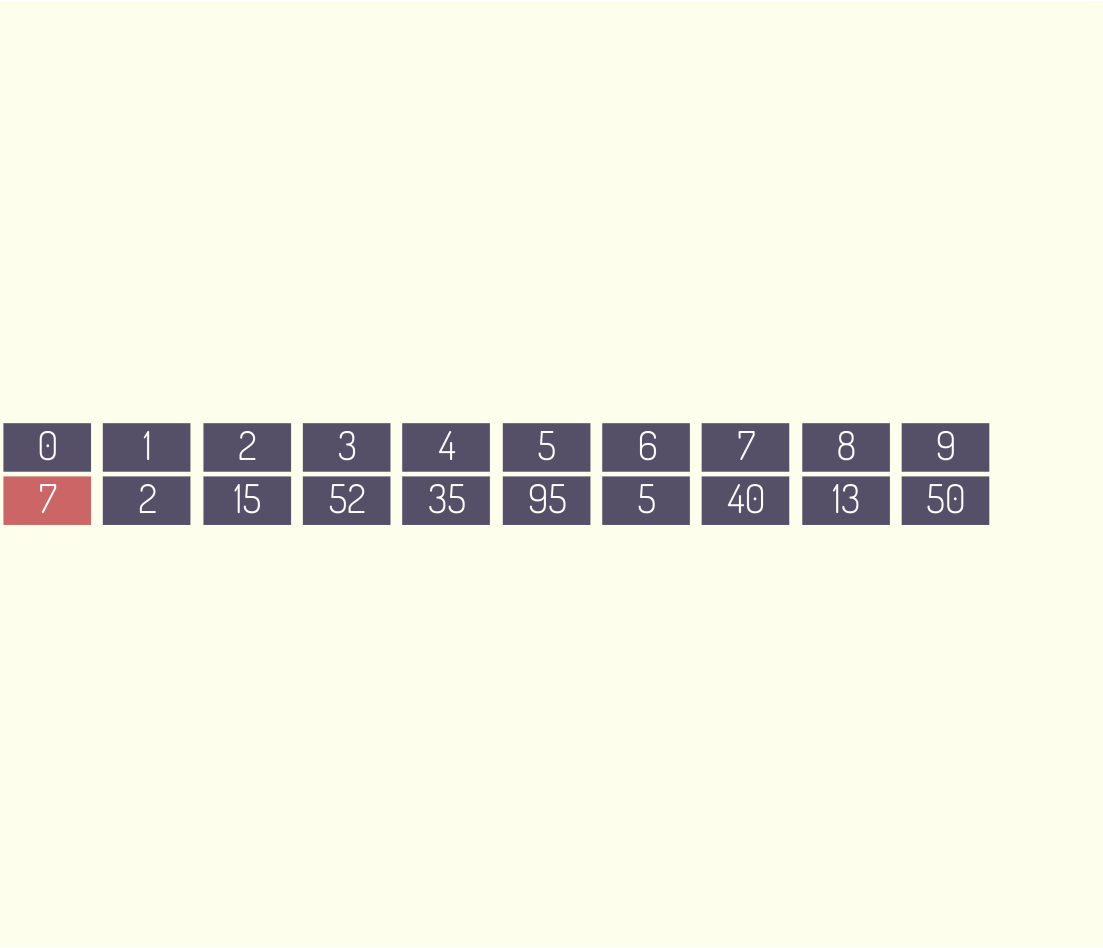
Novamente, o retornou foi o valor de 4,5, dessa forma, como um vetor só possui posições representadas por números inteiros, o resultado é o número 4, sendo assim, a raiz de nossa árvore se iniciará pela quarta posição do vetor, o número 35.

Agora que nossa árvore possui um raiz, vamos começar a adicionar os elementos do vetor na árvore, iniciando pela posição 0 do vetor. Iniciamos então uma comparação do elemento na posição 0 com a raiz, caso este seja maior que o elemento raiz, ele será uma subárvore à direita, caso seja menor, será uma subárvore à esquerda. Vejamos o exemplo a seguir:



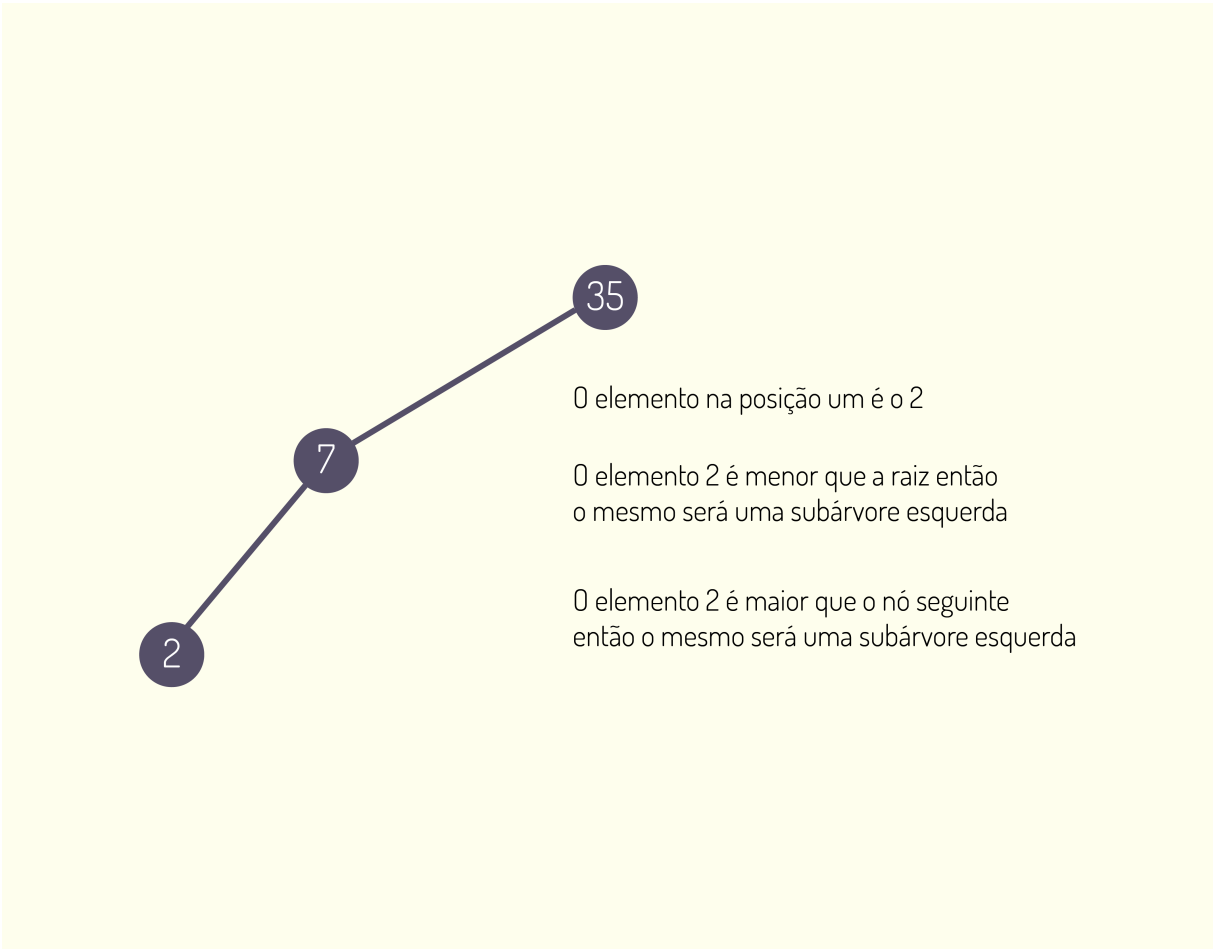
4FIGURA 15.29 - Primeira iteração do vetor para montar a árvore binária FONTE: o autor.

Após alocar o elemento na árvore, passamos para o elemento seguinte no vetor, sendo a posição 1:



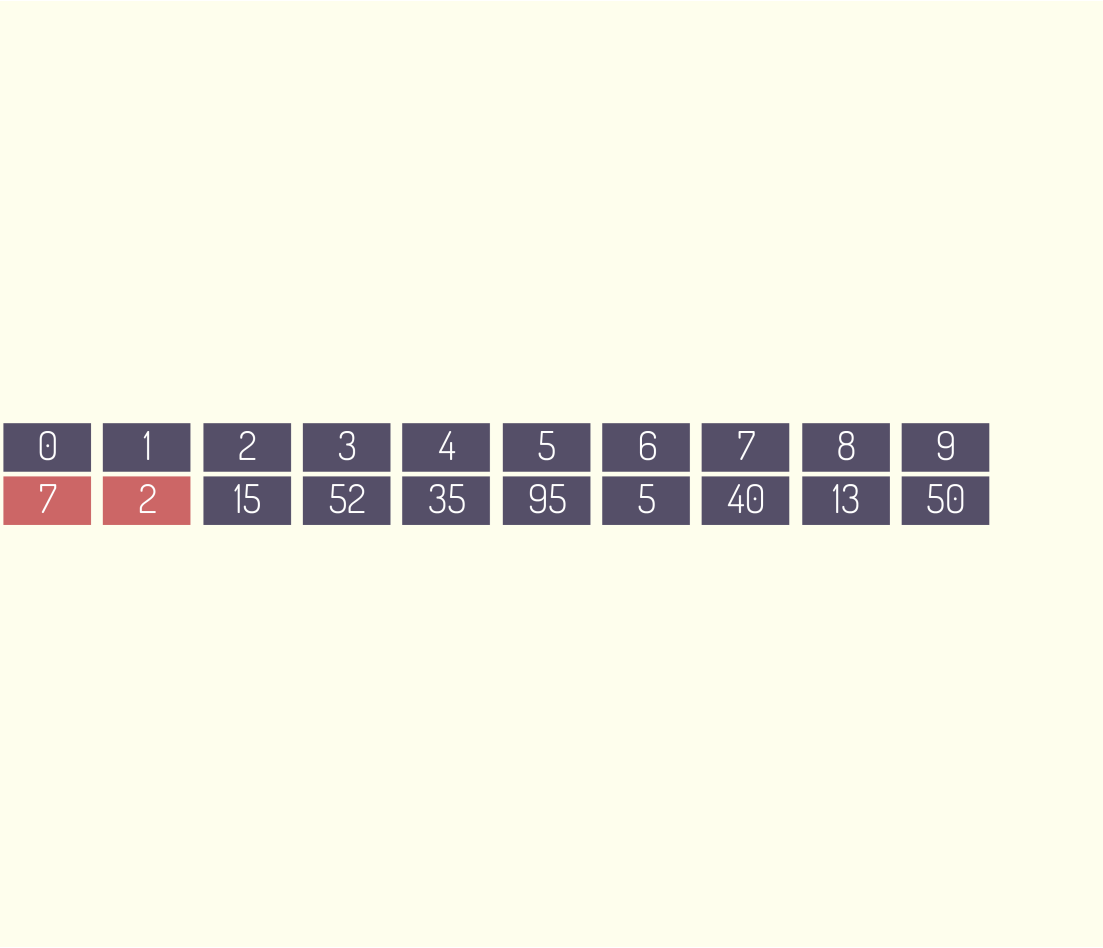
4FIGURA 16.29 - vetor com 10 posições FONTE: o autor.

O elemento alocado nessa posição é o 2, o qual é menor que a raiz, então, é alocado para esquerda, mas, no lado esquerdo, já temos uma subárvore, sendo assim, descemos mais um nível, comparamos com o próximo nó e o inserimos na árvore.



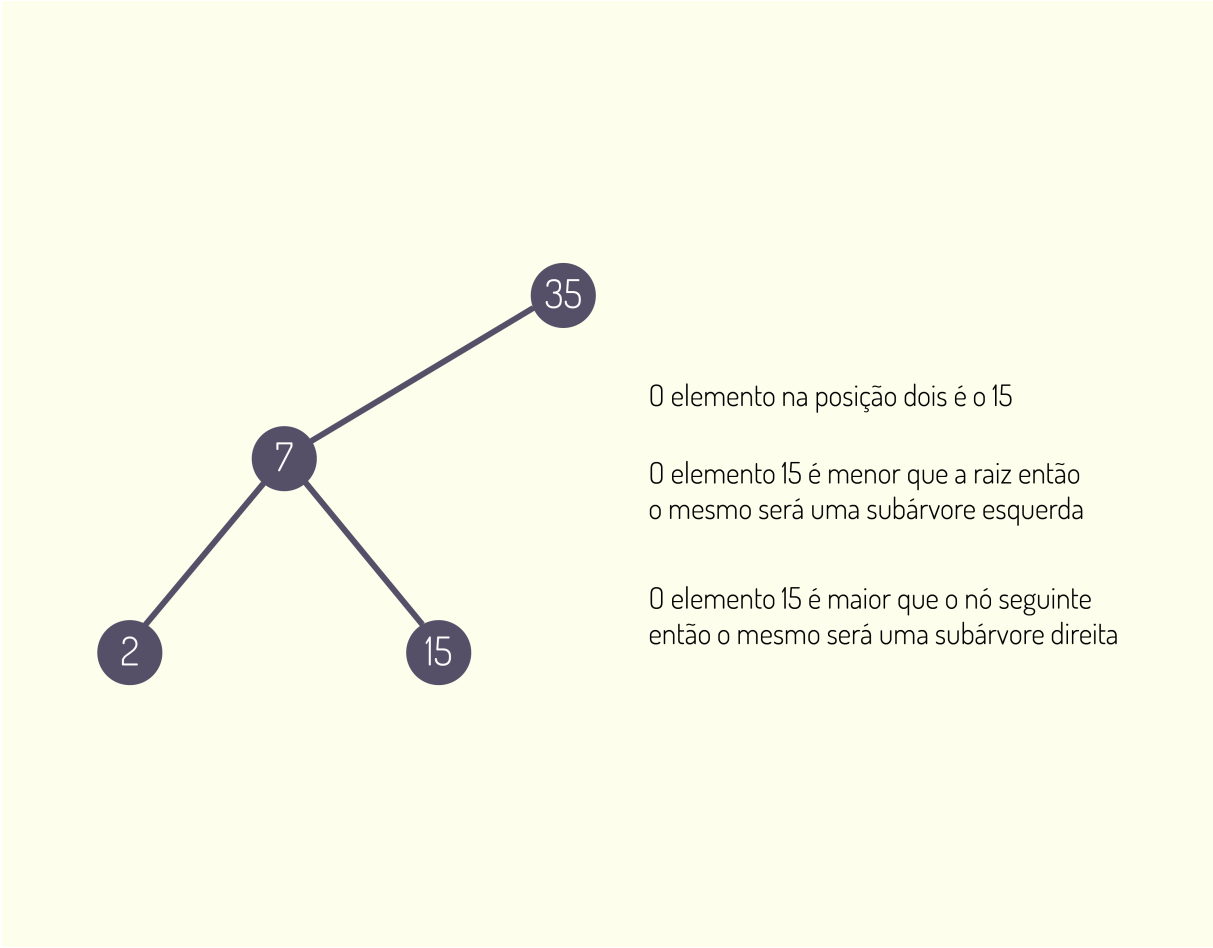
4FIGURA 17.29 - Segunda iteração do vetor para montar a árvore binária FONTE: o autor.

Após alocar o elemento na árvore, passamos para o elemento seguinte no vetor: a posição 2:



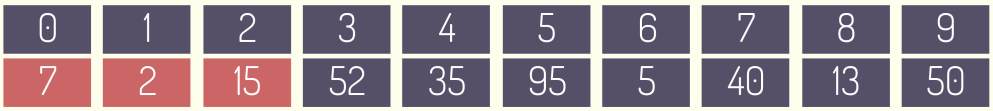
4FIGURA 18.29 - Vetor com 10 posições FONTE: o autor.

Perceba, prezado(a) aluno(a), que o elemento na posição dois do vetor é o 15, que é menor que a raiz, mas maior que o nó seguinte. Nesse caso, vamos inserir o elemento que é um filho à direita do segundo nó:



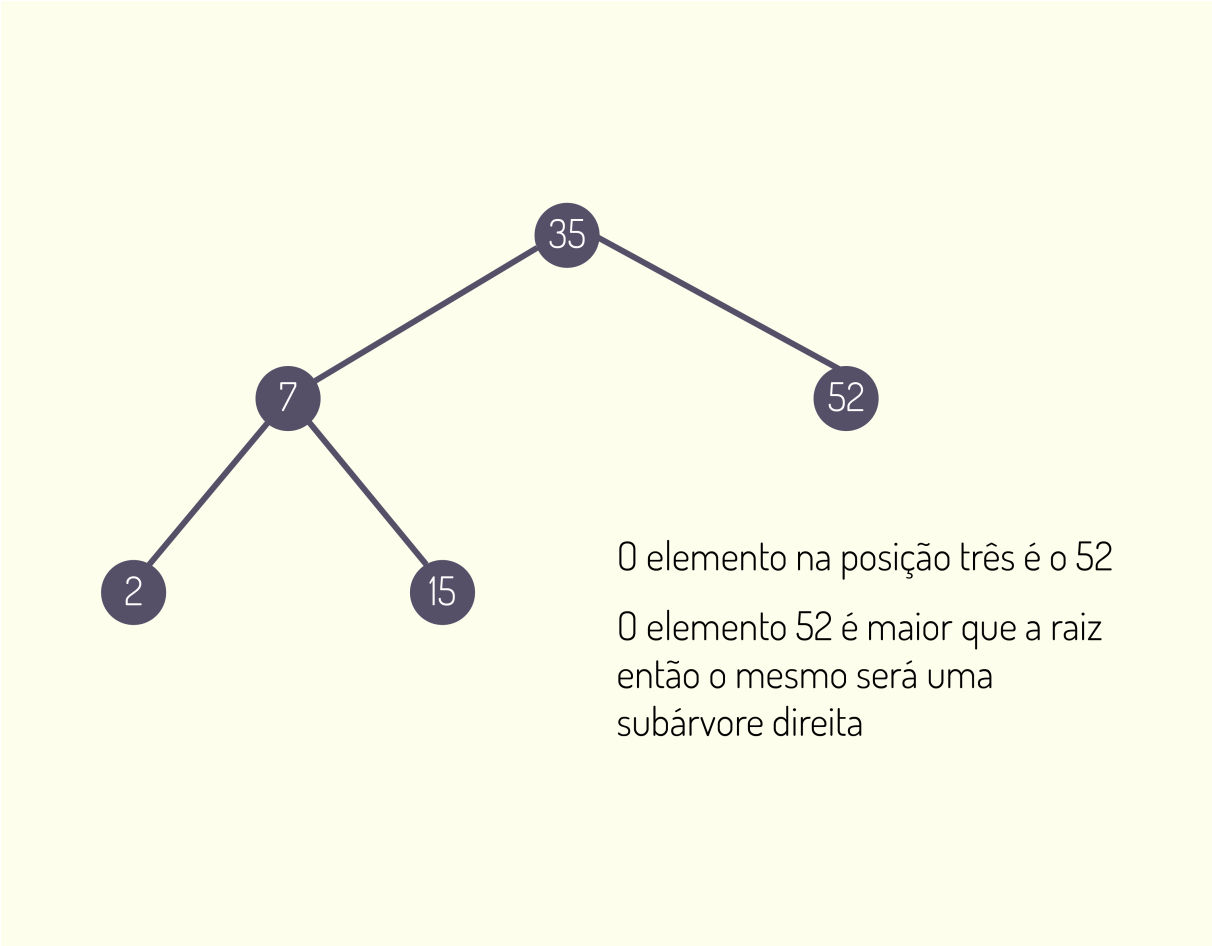
4FIGURA 19.29 - Terceira iteração do vetor para montar a árvore binária FONTE: o autor.

Após alocar o elemento na árvore, passamos para o elemento seguinte no vetor: a posição 3:



4FIGURA 20.29 - Vetor com 10 posições FONTE: o autor.

O elemento na posição três do vetor é o 52, nesse caso, ele é maior que a raiz, assim, é alocado para subárvore direita da raiz:

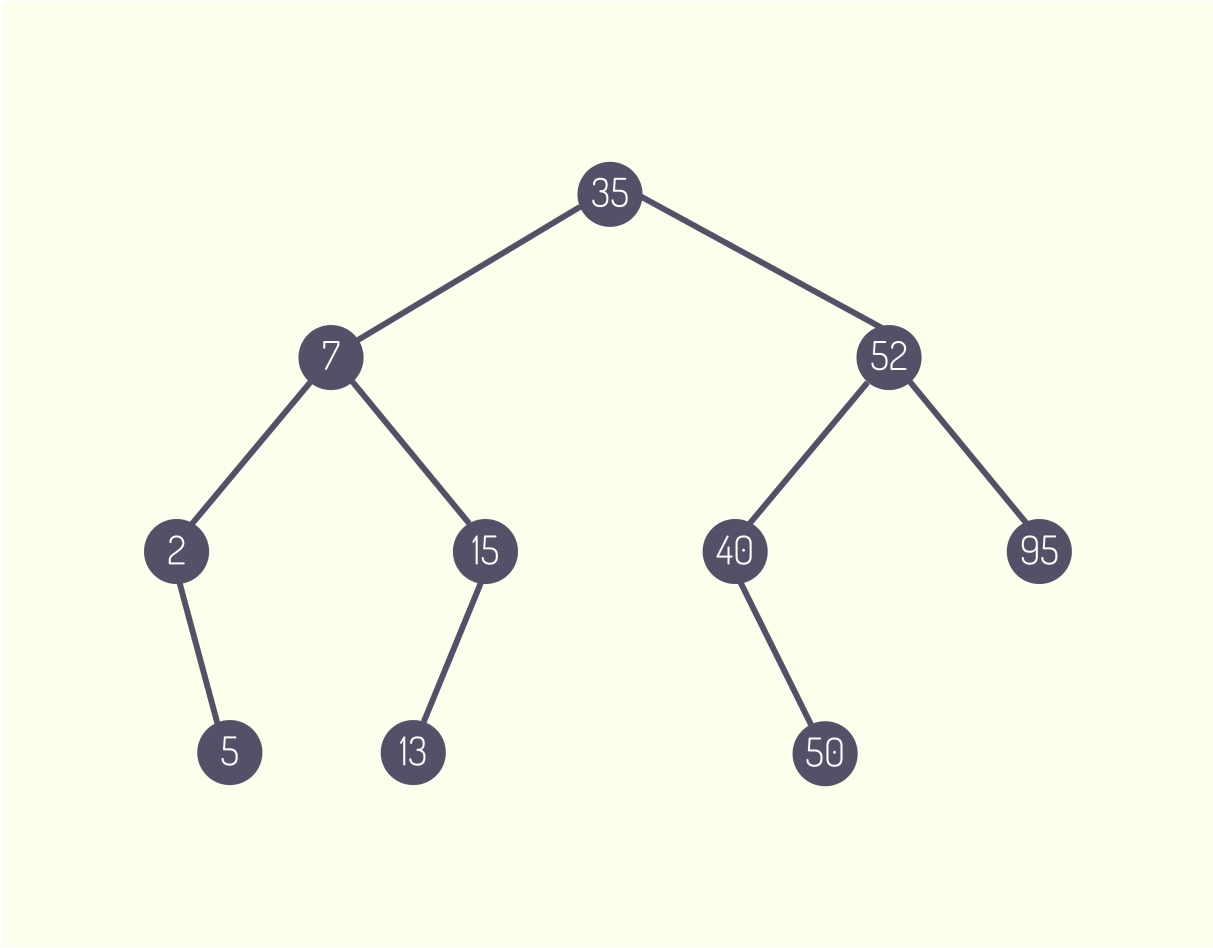


4FIGURA 21.29 - Quarta iteração do vetor para montar a árvore binária FONTE: o autor.

O conceito de comparação dos elementos com a raiz e os níveis é efetuado até que todo o vetor seja percorrido e que todos os elementos estejam na árvore binária. Vejamos como a árvore binária ficará após todos os elementos inseridos.

¶ Fique por dentro

Uma árvore binária é composta por uma raiz, nós e folhas, sendo que os nós da árvore são chamados de pais, cada pai pode possuir no máximo 2 filhos, um à direita e outro à esquerda. Se um nó não possuir nenhum filho, é considerado como uma folha, a cada nó que se percorre na árvore, se percorre um nível da árvore, a qual não tem um limite de níveis.



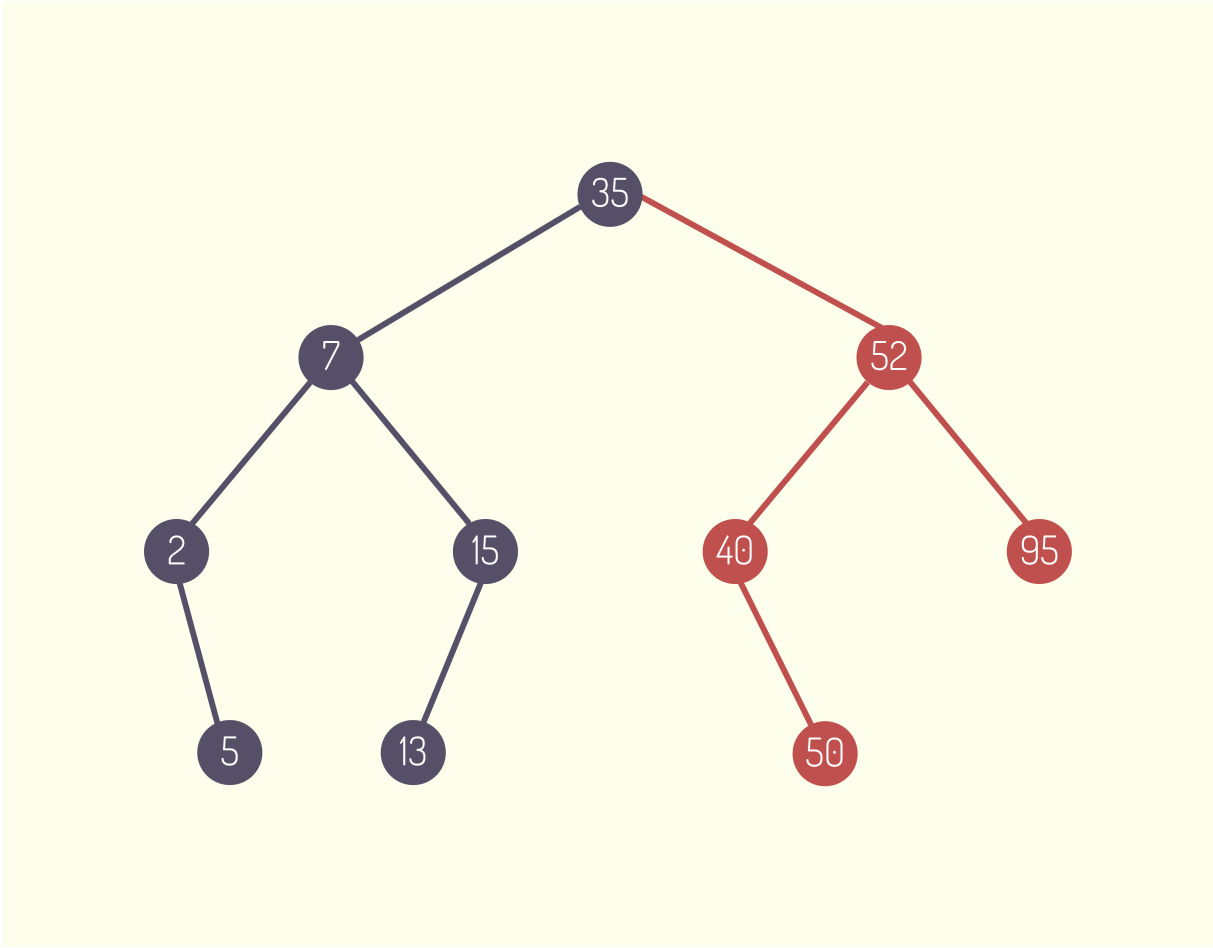
4FIGURA 22.29 - Árvore Binária montada através dos elementos do vetor FONTE: o autor.

É possível identificar que a árvore montada possui três níveis e quatro nós folhas (5, 13, 50, 95), repare, inclusive, que ela não é uma árvore completa ou estritamente binária.

Busca em árvores binárias

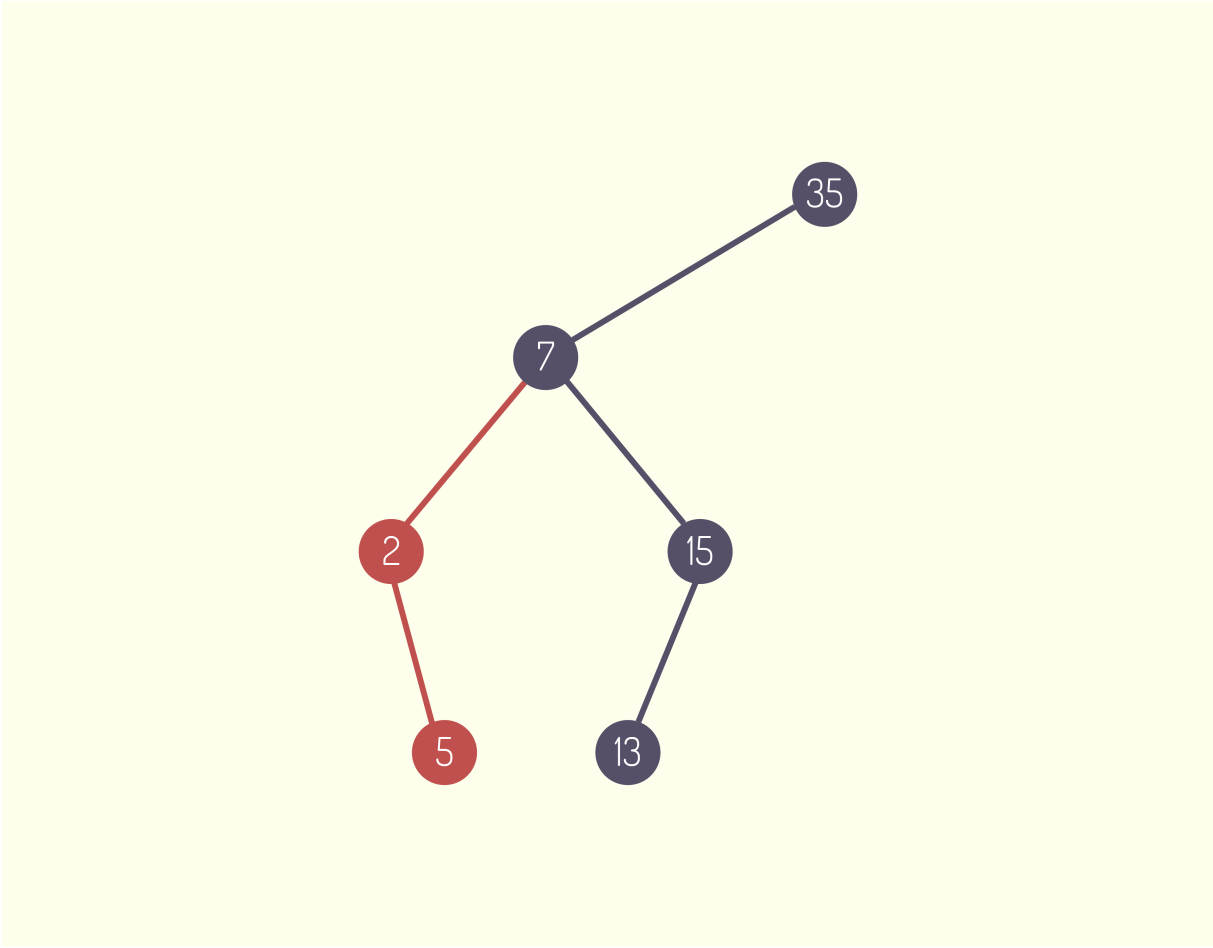
Após montada a nossa árvore binária, vamos aprender como buscar um elemento dentro dessa estrutura. Para simplificar, será utilizada a árvore binária montada anteriormente. O procedimento de busca é bem simples, ele se inicia pela raiz, onde é verificado se o elemento a ser encontrado é maior ou menor que a raiz, caso seja maior que a raiz, todo lado esquerdo é descartado, se não, todo o lado direito é descartado.

O elemento que deverá ser encontrado é o 15, como ele é menor que a raiz, 15 é menor que 35, todo o lado direito é descartado, sendo efetuado o procedimento de busca apenas no lado esquerdo com os números menores, assim reduzimos a complexidade da busca, além de utilizar menos recursos computacionais.



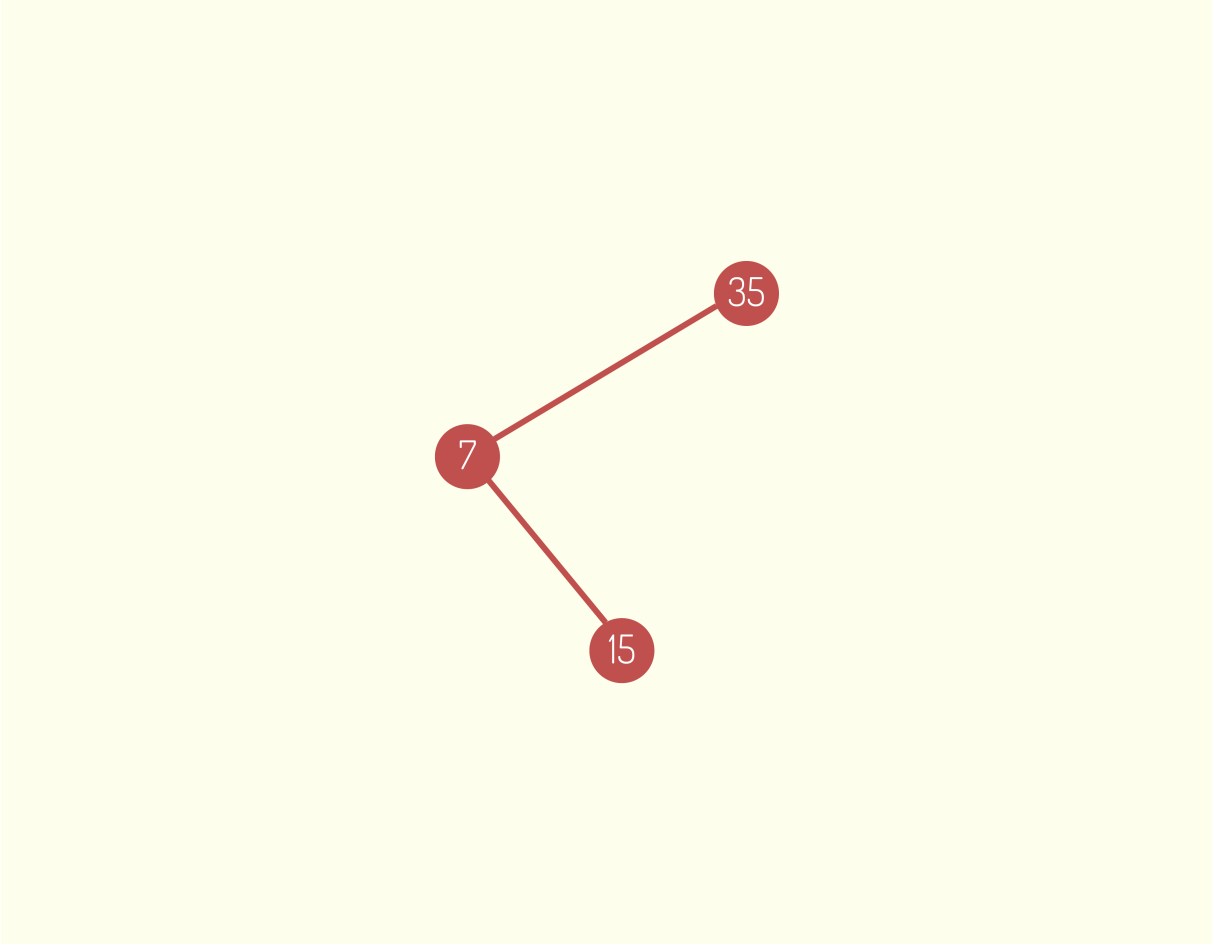
4FIGURA 23.29 - Busca em árvore binária, lado direito removido na primeira iteração da busca FONTE: o autor.

Com a eliminação do lado direito, continuamos a busca descendo pela árvore até o próximo nó, onde se efetua novamente a comparação do elemento que estamos buscando, como 15 é maior que 7, então o lado esquerdo do nó é eliminado também.



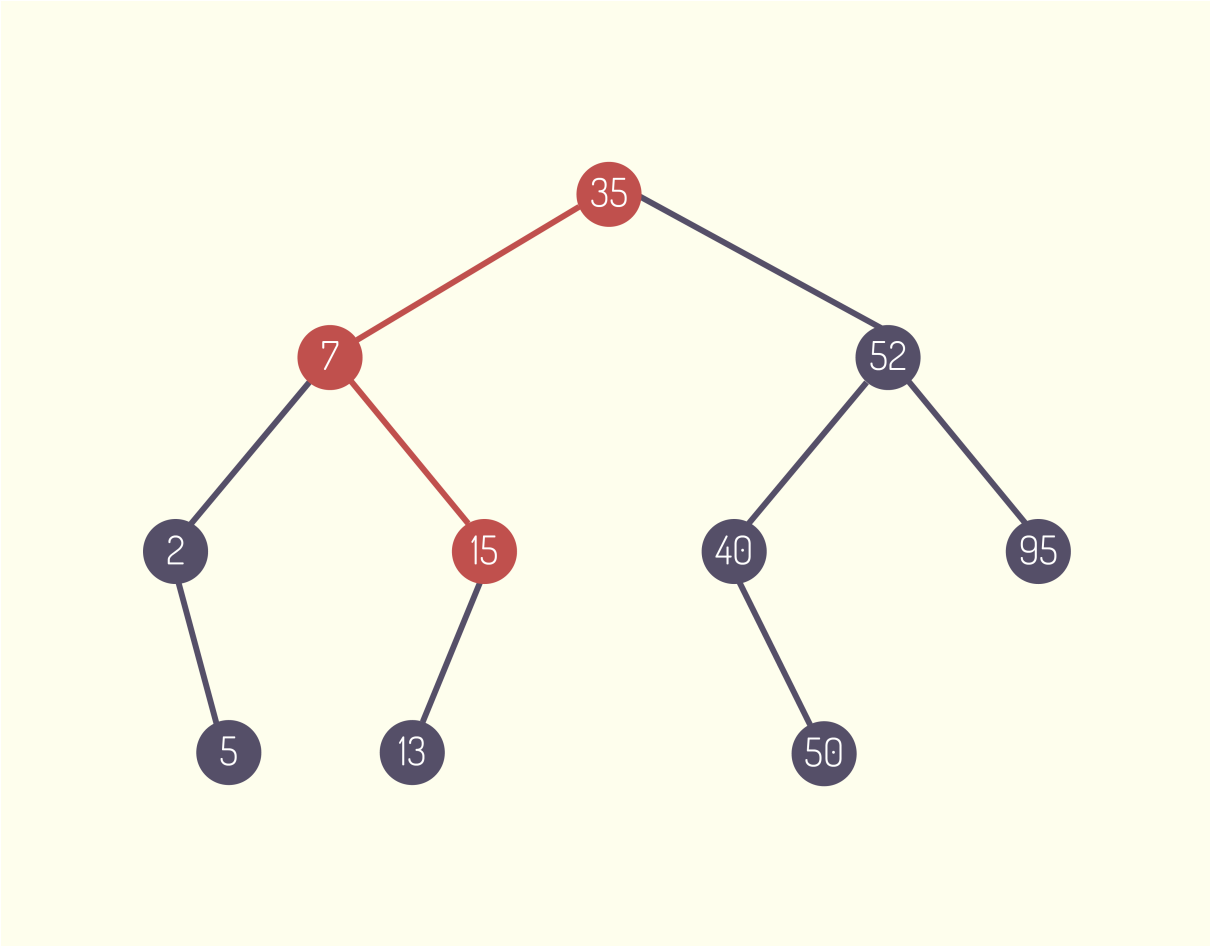
4FIGURA 24.29 - Busca em árvore binária, lado esquerdo removido na segunda iteração FONTE: o autor.

Dando andamento, vamos descer novamente para mais um nível, desta vez, para a direita do nó 7, chegando até o nó 15, onde novamente é efetuada a comparação do elemento buscado com o nó em questão, é identificado que esse foi encontrado, assim, a busca se encerra.



4FIGURA 25.29 - Lado esquerdo é removido na segunda iteração FONTE: o autor.

Por fim, podemos dizer que o caminho percorrido para encontrar o elemento 15 na árvore binária foi 35 -> 7 -> 15, sendo utilizados apenas 3 iterações para encontrá-lo. A busca em árvores binárias pode acontecer nas quais foram montadas a partir de um vetor ou de uma estrutura já pronta.



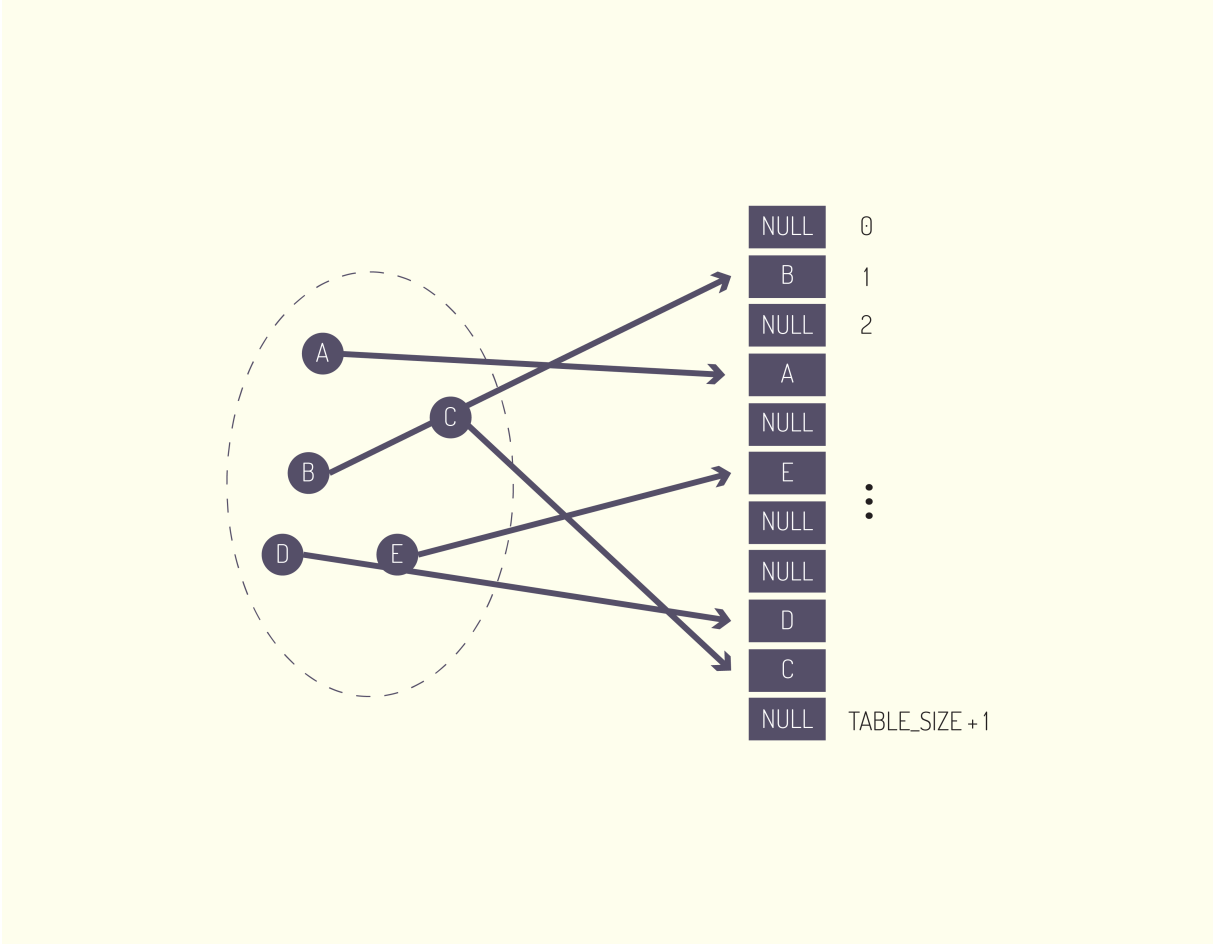
4FIGURA 26.29 - Passos da busca em árvore binária para encontrar o elemento 15 FONTE: o autor.

Busca de dados com tabela Hash

Até o momento, vimos diversas técnicas de busca de dados em vetores ou em árvores binárias. Um conceito que busca melhorar ainda mais a eficiência da busca, diminuindo a sua complexidade, é associar a busca a uma tabela hash conforme veremos.

Tabela Hash

O conceito de tabela de dispersão ou tabela hash tem como função espalhar elementos em uma tabela de forma desordenada, mas como isso pode ser mais eficiente do que os métodos anteriores? Vimos que os métodos de buscas apresentados até o momento tem uma melhor performance com vetores ordenados, a diferença em se utilizar a tabela hash é que ela utiliza uma chave para associar cada elemento ou dados a uma posição na tabela, no caso, um índice da tabela, pode-se dizer que a chave é parte do dado que poderá ser utilizada na pesquisa.



4FIGURA 27.29 - distribuição de elementos em uma tabela Hash FONTE: Pereira (on-line).

Ao utilizarmos esse método, nos deparamos com algumas limitações em seu uso, como a ausência da ordenação dos dados e também o fato de não poder ser varrida para busca simples, sendo utilizado apenas os índices. Vejamos agora como são determinadas as posições dos elementos na tabela hash.

Uma maneira seria aplicar um simples cálculo, o resto da divisão por algum valor, esse valor pode, ser por exemplo, 7, o valor pode variar tanto para mais como para menos, para que você possa compreender melhor, observe a tabela a seguir:

Elemento	Valor	Resto da divisão
15	7	1
25	7	4
14	7	0
85	7	1
9	7	2
75	7	5
80	7	3

Tabela HASH

Índice	Elemento
0	14
1	15
2	9
3	80
4	25
5	75
6	
7	
8	
9	
...	

4FIGURA 28.29 - Alocação dos elementos na tabela hash, utilizando o resto da divisão por 7 FONTE: o autor.

Observe que o resto da divisão é que indica a posição de cada elemento na tabela, mas também veja que temos um problema, pois temos dois elementos a serem alocados na mesma posição:

15 % 7 = 1

85 % 7 = 1

Isso é chamado de colisão, em que temos 1 ou mais elementos a serem alocados na mesma posição, para resolver isso, se cria um vetor na posição onde tivemos a colisão, o ideal é que a tabela hash possua o mínimo de colisões possíveis, assim, não são criados diversos vetores para cada posição, nesse vetor, pode-se ter 2 ou mais elementos, não havendo um limite mínimo ou máximo, é possível tentar resolver o problema de colisões alterando o valor, em que obtemos o resto da divisão.

Elemento	Valor	Resto da divisão
15	7	1
25	7	4
14	7	0
85	7	1
9	7	2
75	7	5
80	7	3

Tabela HASH

Índice	Elemento	
0	14	
1	15	85
2	9	
3	80	
4	25	
5	75	
6		
7		
8		
9		
...		

4FIGURA 29.29 - Colisão de elementos a ser inseridos na mesma posição na tabela hash FONTE: o autor.

Para encontrar qualquer elemento, basta efetuar o mesmo cálculo utilizado para identificar a posição de alocação, assim identificando o índice no qual o elemento está inserido, podendo ser uma chave de vetor ou não. Com isso, a busca sempre retorna à posição exata onde se encontra o elemento buscado, não sendo necessário percorrer o vetor para encontralo.

¶ Fique por dentro

Um site muito conhecido na internet que fala sobre programação e tecnologia da informação é o Viva o linux. Nesse site, é possível tirar dúvidas de informática em geral e também sobre softwares, hardware ou até mesmo programação.

O site disponibiliza diversos scripts prontos, um deles é a implementação da tabela hash em linguagem C. **www.vivaolinux.com.br <https://www.vivaolinux.com.br/script/Tabela-Hash-feita-em-C>** .

Indicação de leitura

Nome do livro: Estrutura de Dados e Técnicas de Programação

Editora: Elsevier

Autor: Dilermando Piva Junior | Gilberto Shigueo Nakamiti | Ricardo Luís de Freitas | Angela de Mendonça Engelbrecht | Francisco Bianchi

ISBN: -10: 85-352-7437-5

ISBN: -13: 978-85-352-7437-0

Este livro traz o conteúdo básico de linguagem de programação, sendo utilizado Algoritmos e Estrutura de dados. Ele trata de Estrutura de Dados e de conceitos e técnicas de programação de Estruturas de Dados em que são apresentados em um formato que prioriza a exposição dos conceitos e problemas de forma clara e objetiva. Para explicar os conceitos por meio de situações-problemas, passa pela resolução da situação, ampliação do tópico, conceituação e aplicação em diversas linguagens de programação. São utilizadas cinco linguagens de programação para as aplicações: algoritmo, pascal, C, Java e Python. Trata-se de um para o ensino-aprendizagem dos tópicos fundamentais de Estrutura de Dados, com potencial utilização em diversas áreas da tecnologia da informação.

Conclusão

Prezado(a) aluno(a)! É com muita satisfação que apresentamos neste material alguns temas sobre estrutura de dados e classificação de dados. O conhecimento sobre estrutura de dados e alguns de seus pontos são fundamentais para o desenvolvimento de softwares e tecnologias em inovação. Esse conhecimento se faz importante em sua vida acadêmica e profissional.

Com essa abordagem, vamos relembrar o que estudamos e os pontos principais do material.

Na unidade I, abordamos inicialmente o conceito de linguagem C, vimos que ela é consideradas a mãe de várias outras linguagens de alto nível como o JAVA, PHP, DELPHI, entre outras. Aprendemos que a linguagem C foi utilizada por muitos anos no desenvolvimento de softwares, por isso, estudamos os seus conceitos básicos e suas principais funcionalidades, o que dá a você conhecimento necessário para desenvolver seus primeiros scripts em linguagem C. Vimos também o que são variáveis, funções e expressões, desvios condicionais simples em compostos (IF e ELSE) e alguns dos tipos de laços de repetição como o FOR, WHILE, DO WHILE, entre diversas funcionalidades.

Na unidade II, conhecemos algumas das formas de estrutura de dados mais utilizadas, como as listas lineares, que podem ser do tipo encadeadas, duplamente encadeadas, circulares, entre outras. Vimos também as estrutra em árvores binárias, uma das formas mais fáceis de se acessar uma informação ou encontrar um elemento. Outras duas estruturas de grande importância são as de Pilha e Fila, também conhecidas como FIFO e LIFO.

Na unidade III, foram trabalhados conceitos mais avançados com a ordenação dos elementos no vetor, utilizando os métodos do BubbleSort (método bolha), MergeSort, QuickSort, entre diversos outros. Apresentamos a você alguns dos mais utilizados, mas ainda temos vários para explorar. Nossa intenção foi fazer com que você vivencie na prática a manipulação dessas estruturas de dados, por isso, sempre buscamos em cada unidade trazer exemplos de implementações dos conteúdo em linguagem C.

Por fim, a unidade IV abordou informações relevantes sobre a busca de elementos em vetores, por intermédio da busca sequencial simples e indexada, sendo utilizando o conceito de busca em binária em vetores, buscando o elemento com base na disposição dos valores no vetor. Foi apresentado também como montar uma árvore binária a partir de um vetor, sendo esse ordenado ou não e também como efetuar a busca em árvores binárias.

Esperamos ter alcançado nosso objetivo em passar nosso conhecimento a você, caro(a) aluno(a). Desejamos que você seja muito feliz ao percorrer o mundo profissional. Muito sucesso e paz!

Referências

ALBANO, Ricardo Sonaglio; ALBANO, Silvie Guedes. **Programação em Linguagem C**. Ed. Ciência Moderna. Rio de Janeiro 2010.

APLICAÇÃO do quicksort num conjunto aleatório de números. Wkimedia Commons, 27 mar. 2009. <https://commons.wikimedia.org/wiki/File:Quicksort-diagram.svg>

ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela Santos de. **Estrutura de dados**: algoritmos, análise de complexidade e implementações em JAVA e C/C++. Pearson Prentice Hall. São Paulo 2010.

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. **Fundamentos da programação de computadores**. 2. ed. Pearson Prentice Hall. São Paulo 2007.

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. **Fundamentos da programação de computadores**: algoritmos, PASCAL, C/C++ (padrão ANSI) e Java. 3. ed. São Paulo: Pearson Education do Brasil, 2012.

ATILHO, Dan. Entendendo pilha e fila. Terminal de informação, 23 jul. 2013. <https://terminaldeinformacao.com/2013/07/23/entendendo-pilha-e-fila/>

COMB SORT. Ashan's Blog, 6 ago. 2015. <http://ashanpeiris.blogspot.com.br/2015/08/comb-sort.html>

DARIUSL. 3d magnifying glass gear blue maschine search loupe. 123RF. <https://br.123rf.com/photo_9529552_3d-magnifying-glass-gear-blue-maschine-search-loupe.html?term=9529552>

DEITEL, Paul; DEITEL, Harvey. C: como programar 6. ed. Pearson Prentice Hall. São Paulo 2011

MIZRAHI, Victorine Viviane. **Treinamento em linguagem C**. Pearson Pretice Hall. São Paulo, 2008.

NORTON, Peter. **Introdução à informática**. Pearson Makron Books. São Paulo 1996

OLIVEIRA, Álvaro B. de; PRADA, Adriana; SILVA, Reginaldo R. da. **Métodos de Ordenação Interna**: Implementação, Análise e Desempenho. Visual Books, 2002

PAPPAS, C. H.; MURRAY, W. H. **Turbo C++ Completo e Total**. São Paulo: Makron, McGraw-Hill, 1991.

PEREIRA, Leinylson Fontinele. Estrutura de Dados Apoio (Tabela Hash). <https://www.slideshare.net/leinylson/estrutura-de-dados-apoio-tabela-hash>

PANYAKHOM, D. Desenho geometria empres. 123RF. <https://br.123rf.com/photo_15396543_desenho-geometria-empres.html?term=15396543>

PEREIRA, Silvio do Lago. Linguagem C. <https://www.ime.usp.br/~slago/slago-C.pdf>

PUGA, Sandra; RISSETTI, Gerson. **Lógica de programação e estrutura de dados, com aplicações em JAVA**. 2. ed. Pearson Prentice Hall. São Paulo 2009.

RANGEL, W. Celes e J. L. Listas Encadeadas. <http://www.ic.unicamp.br/~ra069320/PED/MC102/1s2008/Apostilas/Cap10.pdf>

RONSTIK. Programador ocupação - escrever código de programação no laptop. 123RF. <https://br.123rf.com/photo_35650670_programador-ocupa%C3%A7%C3%A3o---escrever-c%C3%B3digo-de-programa%C3%A7%C3%A3o-no-laptop.html?term=35650670>

SCHILDT, Herbert. **C Completo e Total**: 3. edição revisada e atualizada. Pearson Makron Books, São Paulo, 1997.

SCHILDT, Herbert. **Linguagem C**: Guia do Usuário. McGraw-Hill, São Paulo, 1986.

TENENBAUM, Aaron M. *et al.* **Estrutura de dados usando C**. Pearson Makron Books. São Paulo 1995

Atividades

Atividades - Unidade I

As entradas de dados são informações atribuídas pelo usuário, é reservado um espaço na memória por intermédio da declaração de variáveis. A linguagem C possui tipos específicos como char, int, float, entre outros, se a variável for declarada de modo errado, o programa apresentará erros. Sobre a forma de declarar as variáveis, assinale a alternativa incorreta de como efetuar esse processo.

A) char nome, telefone; float altura1; int numero1.

B) ="alternative-true">int numero, 1lugar; float peso; char nome completo, preço.

C) float peso, altura; char nome[20], curso[20]; int rg, cpf, codigo_pessoa.

D) float Preco, ValorTotal; int QTDETotal, Unidade; char Cliente, fornecedor.

E) int posicao; char local; float comprimento.

As instruções condicionais tem como objetivo tomar uma decisão caso a condição seja verdadeira, para testar a condição, pode-se utilizar expressões ou operadores. Assinale a alternativa correta sobre a instrução condicional IF:

A) ="alternative-true">if (i >= j) { printf(" 1º número é maior que o 2º "); }

B) if ( ) k == o { printf(" Produto já existente"); }

C) if (nome1 igual nome2) { printf(" cliente já cadastrado "); }

D) if (celular != telefone && comercial != telefone) { } printf("telefones diferentes");

E) if (opcao1 == opcao2 and opcao3 != opcao1) { prinf("opções diferentes); }

Podemos agrupar variáveis do mesmo tipo através de um vetor, sabendo que este armazena os dados de modo linear, em formar de linha, ou em uma matriz, utilizando-se linhas e colunas. Assinale a alternativa correspondente a onde os dados do Aluno Carlos estão sendo armazenados: nome[15], dados_pessoa[5][6], nota[1]:

A) O nome está na 15ª posição, os dados pessoais estão na posição 56 da matriz, sua nota está na 1ª posição

B) ="alternative-true">O nome se encontra na 15ª posição, os dados do aluno estão na linha 5 e coluna 6, e sua nota está na 1ª posição.

C) O nome está na 15ª posição, os dados pessoais estão na coluna 5 linha 6 da matriz, sua nota está na 1ª posição.

D) O nome está na 15ª posição, os dados estão na posição 56, a nota do aluno é 1.

E) Nenhuma das alternativas, pois a forma de declarar os vetores e a matriz está incorreta.

O principal benefício de se utilizar uma função no desenvolvimento de um programa é o fato de se reaproveitar trechos do código e diminuí-lo. Para manipular os dados dentro de uma função, podem-se usar variáveis. Com base nisso, assinale a alternativa correta sobre os tipos de declarações de uma variável em linguagem Cl

A) Locais, externas, parâmetros gerais.

B) ="alternative-true">Parâmetros formais, locais, globais.

C) Globais, internas, gerais.

D) Parâmetros formais, globais, internas.

E) Globais, locais, independentes.

Atividades - Unidade II

O mesmo número pode ter um significado diferente ao ser convertido para algum sistema, pensando nisso, qual será o valor de 11 convertido para hexadecimal? Assinale a alternativa correta.

A) Equivalente a 10.

B) ="alternative-true">Equivalente a 17.

C) Equivalente a 11.

D) Equivalente a 1.

E) Nenhuma das alternativas.

A estrutura de dados em lista permite armazenar elementos em um vetor, com ela é possível organizar os elementos de forma crescente ou decrescente na lista. Em alguns casos, ao percorrer a lista, podemos nos movimentar para frente e para trás. De qual tipo de lista estamos falando?

A) Lista encadeada simples e circular.

B) ="alternative-true">Lista duplamente ordenada.

C) Lista encadeada simples.

D) Lista duplamente modular.

E) Lista circular simples

Uma árvore binária é um tipo de estrutura de dados utilizada para organização das informações, sendo composta pela raiz, nós e folhas. Cada nó pode ter no máximo 2 filhos, e cada filho, um único pai, exceto a raiz, que não possui pai. Referente às árvores binárias, assinale a alternativa que descreve uma árvore estritamente binária.

A) Árvore que possua no máximo um filho por pai.

B) ="alternative-true">Árvore na qual todo nó que não é folha tem sempre duas subárvores esquerda e direita não vazias.

C) Árvore que possui somente o lado esquerdo.

D) Árvores que são formadas apenas por folhas.

E) Árvore que tenha vários nós à esquerda e somente uma folha à direita.

Imagine que você trabalha na área da construção civil, existem vários tijolos espalhados que precisam ser organizados, cada tijolo foi colocado em cima do outro até atingir o número de 50 tijolos. Conforme a demanda da obra, você foi retirando os tijolos e, sem perceber, você estava aplicando um tipo de estrutura de dados. Qual dos tipos a seguir representa os processos dessa organização de tijolos?

A) Filas.

B) ="alternative-true">Pilhas.

C) Listas lineares.

D) Árvores binárias.

E) Conjuntos

Atividades - Unidade III

Ricardo trabalha no almoxarifado de um empresa que efetua distribuição de alimentos, ao chegar novos produtos, ele deve colocá-los em ordem através da data de validade. Em uma fila de pacotes, ele compara o atual com o seguinte, caso a data de validade do atual for maior, ele troca os pacotes de lugar; ele efetua essa comparação até organizar toda a fila. Qual o método utilizado por Ricardo para efetuar a ordem dos pacotes.

A) InsertionSort.

B) ="alternative-true">BubbleSort.

C) Troca e inserção.

D) Dois pares Sort.

E) TrocaSort.

Almir joga futebol com seus amigos todos os finais de semana e sempre tem problemas na hora de separar os times. Ele lembrou de um método de ordenação que está aprendendo na faculdade e decidiu utilizá-lo. Quem tivesse a idade menor que a dele ia para o time da esquerda, os demais com idade superior a dele para a direita. Qual algoritmo ele está utilizando?

A) DivisionSort.

B) ="alternative-true">QuickSort.

C) SelectionSort.

D) OrderSort.

E) AlterSort.

Nelson é um pai dedicado e está sempre ajudando sua filha em seus estudos. Ao estudarem sobre as guerras da antiguidade, percebeu que alguns dos grandes líderes se utilizavam de um dos métodos de ordenação que estudou na faculdade, o qual era conhecido por dividir para conquistar. De qual algoritmo ele está se lembrando?

A) MergeConquista.

B) ="alternative-true">MergeSort.

C) Bubblesort.

D) ShellSort.

E) SeparaSort.

O CombSort foi desenvolvido para solucionar um problema encontrado na ordenação dos vetores, os chamados turtles, o objetivo era posicionar de modo rápido esses elementos para início do vetor, para isso, o algoritmo se utiliza de duas técnicas, GAP e trocas, quais são os métodos de ordenação que também se utilizam dessas técnicas.

A) BubbleSort e InsertionSort.

B) ="alternative-true">ShellSort e BubbleSort.

C) QuickSort e TrocaSort.

D) BubbleSort e MergeSort.

E) TurtleSort e QuickSort.

Atividades - Unidade IV

Uma busca sequencial é utilizada para percorrer um vetor ou lista para encontrar um elemento. Seu método é bem simples, mas muito eficaz. Já a busca sequencial indexada tem a proposta de melhorar a busca apenas sequencial. Quais são as principais diferenças entre os dois tipos de busca.

A) A busca apenas sequencial só percorre vetores ordenados. A busca sequencial só é aplicada em vetores desordenados.

B) ="alternative-true">A busca sequencial pode varrer vetores e listas ordenados ou não. A busca sequencial indexada se utiliza de uma tabela extra para efetuar a busca.

C) A única diferença é que a busca sequencial pode-se utilizar de vetores ordenados ou não, já a busca indexada, apenas de vetores ordenados

D) A busca indexada se utiliza de um vetor extra chamado de tabela auxiliar, a busca sequencial não se utiliza de vetores extra.

E) A busca sequencial se utiliza de uma tabela adicional, a indexada, não.

Para que as buscas binárias e por interpolação sejam utilizadas, os vetores devem estar em um determinado estado, após isso, é possível verificar se o vetor está de maneira uniforme ou não. O que se deve fazer para utilizar as buscas binárias ou interpolação? Caso o vetor não esteja uniforme, qual tipo de busca é indicado?

A) Os vetores devem estar separados em sublistas. A busca indicada é a binária.

B) ="alternative-true">Os vetores devem estar ordenados. O método indicado é a busca binária.

C) Os vetores devem estar ordenados. A busca indicada para o procedimento é a por interpolação.

D) Os vetores devem ser listas lineares. O método indicado é a busca binária.

E) Vetores desordenados. Ambas as buscas podem ser utilizadas.

A busca binária pode se utilizar de uma estrutura já criada ou pode-se montar uma nova árvore a partir de um vetor, em que, através de um cálculo simples, encontramos o meio do vetor. Sobre o vetor a ser utilizado para montar a árvore binária, é correto afirmar:

A) Só é possível montar uma árvore binária se o vetor estiver ordenado.

B) ="alternative-true">O vetor pode estar ordenado ou não ordenado.

C) Árvores binárias só podem ser montadas com vetores desordenados.

D) Só é possível montar uma árvore com números pares.

E) Não é possível montar árvores binárias com vetores, apenas com listas.

As tabelas de dispersão são muito interessantes de se trabalhar. Vimos que, para alocar os elementos, efetuamos um cálculo bem simples, e o resto da divisão será a posição onde o elemento será alocado na tabela hash, em algumas situações, o resto da divisão é o mesmo para mais que um elemento. Qual o nome desta situação e como é possível resolver ?

A) Conflito de elementos, e é resolvido criando-se um vetor.

B) ="alternative-true">Colisão, para resolver, cria-se um vetor na posição de alocação.

C) Colisão, e se resolve criando uma tabela auxiliar.

D) Colisão e, para resolver, se apresenta o erro no método.

E) Equivalência de elementos, e se resolve criando um vetor na posição de alocação.