

Estrutura de Dados I

2025/2

Nicolas Ramos Carreira

Sumário

1	Intuito	4
2	Fundamentos em C	5
2.1	Sobre a linguagem C	5
2.2	Estrutura de um programa em C	5
2.3	Aspectos da linguagem C	6
2.3.1	Variaveis	6
2.3.2	Tipos de dados	7
2.3.3	Input e output	8
2.3.4	Contantes	11
2.3.5	Operadores	12
2.3.6	Coerção de tipos	14
2.3.7	Condicionais	14
2.3.8	Loops	16
2.3.9	Arrays	16
2.3.10	Struct - Criação de tipos	18
2.3.11	Comando typedef	18
3	Acerca de ponteiros	19
3.1	O porquê de estudar esse topico	19
3.2	O que são e como usá-los	19
3.2.1	O que é	19
3.2.2	Declaração de ponteiros	19
3.2.3	Detalhe após a declaração	20
3.2.4	Exemplo de uso	20
4	Falando de funções	22
4.1	O que é	22
4.2	Estrutura	22
4.2.1	Corpo	23
4.2.2	Parametros	23
4.2.3	Retorno	23
4.2.4	Escopo	24
4.3	Passagem de parâmetros	24

4.3.1	Passagem por valor	24
4.3.2	Passagem por referência	24
4.3.3	Arrays como parametro	25
4.3.4	Struct como parametro	25
4.4	Recursão	25
5	Introdução aos algoritmos	26
5.1	Algoritmos de busca	26
5.1.1	Busca linear - Não ordenada	26
5.1.2	Busca linear - Ordenada	27
5.1.3	Busca Binária	28
5.1.4	Busca em array de struct	29
5.2	Algoritmos de ordenação	30
5.2.1	Bubblesort	30
5.2.2	Insertion sort	35
5.2.3	Selection sort	35
5.2.4	Merge sort	36
5.2.5	Shell sort	36
5.2.6	Quicksort	36
6	Alocação dinâmica	38
6.1	Conceito	38
6.2	Tipos de alocação dinamica	39
6.2.1	malloc	39
6.2.2	calloc	40
6.2.3	realloc	40
6.2.4	free	41
6.3	Casos especiais da alocação dinamica	42
6.3.1	Arrays	42
6.3.2	Structs	45
7	Lista encadeada, Pilha e Fila	47
7.1	Listas encadeadas	47
7.1.1	Como funciona?	47
7.1.2	Implementação	48
7.2	Pilha	51
7.2.1	Como funciona	51
7.2.2	Curiosidade	51
7.2.3	Implementação	52
7.3	Fila	54
7.3.1	Como funciona	54
7.3.2	Curiosidade	54
7.3.3	Implementação	55

8 Arvores	58
8.1 Conceituando	58
8.1.1 Detalhando a estrutura	59
8.1.2 Curiosidade	60
8.2 Arvores binárias	60
8.2.1 Conceito	60
8.2.2 Propriedades	62
8.3 Arvore binaria de busca	64
8.3.1 Operações básicas	65
8.4 Arvores AVL	67
8.4.1 Conceito	67
8.4.2 Funcionamento	68
8.4.3 Rotações	68
9 Tabela Hash	76
9.1 Conceito	76
9.2 Funcionamento	77
9.3 As colisões	77

Capítulo 1

Intuito

O intuito deste documento é documentar o meu aprendizado da disciplina de estrutura de dados 1. Nesta disciplina começamos estudando sobre a linguagem C até entrar nas principais estruturas de dados.

Capítulo 2

Fundamentos em C

2.1 Sobre a linguagem C

A linguagem C é uma das linguagens de programação mais influentes e utilizadas da história da computação. Criada na década de 1970 por Dennis Ritchie nos laboratórios Bell, ela foi projetada para ser uma linguagem de propósito geral, eficiente e próxima do hardware, permitindo alto desempenho.

C é considerada uma linguagem de médio nível, pois combina características de linguagens de baixo nível (como manipulação direta de memória) com recursos de alto nível (como funções e estruturas). Sua sintaxe influenciou muitas outras linguagens modernas, como C++, Java, Csharp e até mesmo Python em alguns aspectos.

É amplamente usada em sistemas operacionais, softwares embarcados, drivers e aplicações que exigem alto desempenho. Além disso, aprender C é um ótimo ponto de partida para entender conceitos fundamentais de programação e arquitetura de computadores.

2.2 Estrutura de um programa em C

```
● ● ●

#include <stdio.h>

int main(){
    printf("Hello world!\n");
    return 0;
}
```

A imagem acima mostra um programinha extremamente simples em C, um Hello, world. Para iniciar um programa em C, nós sempre começamos declarando a biblioteca principal, que é a stdio.h (poderíamos ter outras bibliotecas inclusive, mas essa é a principal e DEVE estar lá).

Depois disso, nós declaramos o local do programa principal, onde fazemos o programa em si.

Um detalhe é que ao final de cada coisa SEMPRE temos que ter o ponto e vírgula (;), pois se não o nosso programa não compila.

2.3 Aspectos da linguagem C

2.3.1 Variaveis

O que são e pra que são usadas

Varivel, em linguagens de programação, é basicamente uma posição alocada da memória para guardar uma informação. Variaveis podem ser modificadas pelo programa e devem ser definidas antes de ser utilizadas

Declaração de variaveis em C

Para definir variaveis em C, nós precisamos passar o tipo de dado e nome da variavel, no formato:

<tipo de dado> nome-da-variavel;

Obs: ao fazer da forma acima, estamos apenas declarando a variavel, sem atribuir um valor a ela

O tipo de dado deve ser aqueles que são aceitos pela linguagem (inteiro, decimais, caracteres, booleanos..), mas como falaremos sobre tipos de dados mais pra frente, não entraremos em detalhes agora. O nome da variavel é algo bem importante a se considerar, pois existem algumas regras e boas práticas importantes quanto a isso:

- Nomes de variaveis devem iniciar com letras ou underscore
- Os caracteres da variavel devem ser letras, numeros ou underscore (não utilizar acentos ou simbolos)
- Não utilizar espaço em nomes de variaveis
- Palavras chaves (palavras que são reservadas pela linguagem para fazer determinadas coisas) não podem ser usadas como nomes
- Letras maiusculas e minusculas são consideradas diferentes

Só para deixar totalmente claro, as palavras chaves que a linguagem C usa são:

auto	break	case	char	const	continue	do	double
else	for	int	union	static	default	void	return
enum	goto	long	unsigned	struct	extern	while	sizeof
float	if	short	volatile	switch	register	typeof	

Atribuição de valores em variaveis

Tendo o formato <tipo de dado> nome-da-variavel, podemos atribuir valores a elas (ou seja, armazenar valores dentro da memória). Para isso, basta fazer:

<tipo de dados> nome-variavel = valor;

2.3.2 Tipos de dados

Como falamos anteriormente na parte de variaveis, quando vamos defini-las, nós temos que declarar o tipo de dado da variavel. O tipo de dado define os valores que aquela variavel pode assumir e as operações que podem ser realizadas com ela. Os tipos de dados principais são: char, int, float e double

Char

Um byte que armazena o código de um caractere do conjunto de caracteres local

Int

Um inteiro cujo o tamanho do numero que pode ser alcançado depende do processador (tipicamente 16 ou 32 bits)

Float

Basicamente numeros decimais com precisão simples (em C a parte decimal usa ponto e não vírgula)

Double

Também números decimais, mas com precisão dupla. É usados para numeros muito pequenos (científicos por exemplos) ou muito grandes

Bool

Esse tipo de dados é muito interessante, pois ele pode assumir dois valores: verdadeiro ou false (true ou false). Em outras linguagens, nós temos literalmente um valor True e False. No entanto, na linguagem C nós não temos True e False, mas podemos representá-los como 1 e 0, respectivamente.

Outros tipos

Na imagem abaixo, você poderá ver alguns outros que são utilizados:

Tipo	Bits	Intervalo de valores
char	8	-128 A 127
unsigned char	8	0 A 255
signed char	8	-128 A 127
int	32	-2.147.483.648 A 2.147.483.647
unsigned int	32	0 A 4.294.967.295
signed int	32	-32.768 A 32.767
short int	16	-32.768 A 32.767
unsigned short int	16	0 A 65.535
signed short int	16	-32.768 A 32.767
long int	32	-2.147.483.648 A 2.147.483.647
unsigned long int	32	0 A 4.294.967.295
signed long int	32	-2.147.483.648 A 2.147.483.647
float	32	1,175494E-038 A 3,402823E+038
double	64	2,225074E-308 A 1,797693E+308
long double	96	3,4E-4932 A 3,4E+4932

2.3.3 Input e output

Input e output é basicamente a entrada e a saída de dados. As vezes, podemos querer receber do usuário alguns valores, para fazer alguma coisa com eles e depois entregá-los com modificações. É basicamente isso. Um detalhe é que para o output, não necessariamente nós precisamos ter recebido algo.

Especificadores de formato

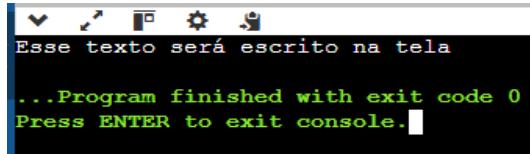
Saída com printf()

Vamos começar com a saída de dados. Para exibir algo na tela. Fazemos:



```
#include <stdio.h>
int main(){
    printf("Esse texto será escrito na tela");
    return 0;
}
```

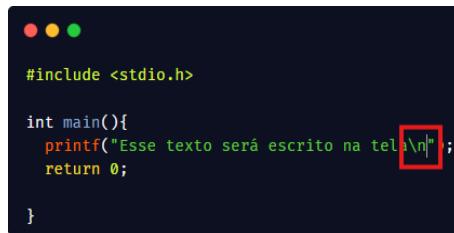
Ao fazer isso, em nosso terminal será exibido o texto que digitamos dentro do printf ("Esse texto será escrito na tela"). Veja:



```
Esse texto será escrito na tela
...Program finished with exit code 0
Press ENTER to exit console.
```

Uso do escape no printf()

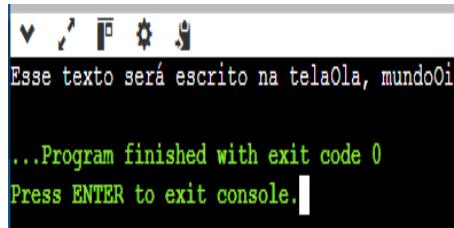
Um detalhe é que algo que podemos utilizar no printf é caractere de escape . Esse caractere é utilizado sempre ao final do que queremos escrever na saída e ele serve para quebrar a linha após a saída. Veja:



```
#include <stdio.h>

int main(){
    printf("Esse texto será escrito na tela\n");
    return 0;
}
```

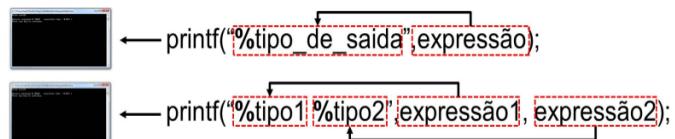
Se fizermos vários printf, por exemplo, e não usarmos o caractere de escape em nenhum deles, o que escrevemos nos prints, ficará tudo junto. Veja:



```
Esse texto será escrito na telaOla, mundoOl
...Program finished with exit code 0
Press ENTER to exit console.
```

Exibindo valores de variaveis no output

Se quisermos que em nosso output seja usada alguma variável, temos que utilizar o seguinte formato:



Isso acima significa que se quisermos passar no output uma variável que tenha o tipo int, nós teríamos que passar o tipo de saída dentro das aspas duplas e depois separar por vírgula passando a nossa variável. Mas você deve

estar se perguntando: Como assim tipos de saída? Veja abaixo os tipos de saída que usaremos no output (printf):

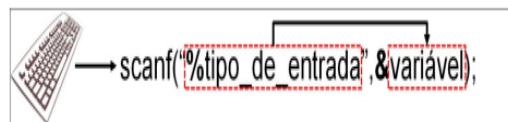
Alguns tipos de saída	
%c	escrita de um caractere (char)
%d ou %i	escrita de números inteiros (int ou char)
%u	escrita de números inteiros sem sinal (unsigned)
%f	escrita de número reais (float ou double)
%s	escrita de vários caracteres
%p	escrita de um endereço de memória
%e ou %E	escrita em notação científica

Ou seja, seguindo o exemplo da variável de tipo int que tínhamos dado, se quiséssemos exibi-la no output (printf), faríamos:

```
printf("%d", variavel);
```

Entrada com scanf()

Agora, falando sobre entrada de dados, o comando que utilizamos para passar dados para o nosso programa é o scanf(). Esse comando permite realizar a leitura de dados da entrada padrão (teclado). Sua estrutura é a seguinte:



Sendo que, os tipos de entrada são praticamente os mesmos que vimos nos tipos de saída. Veja:

Alguns tipos de saída	
%c	leitura de um caractere (char)
%d ou %i	leitura de números inteiros (int ou char)
%f	leitura de número reais (float ou double)
%s	leitura de vários caracteres

Podemos ainda realizar a leitura de mais de um valor (assim como podemos fazer o output de mais de um valor). É bem parecido com o output também. Veja:

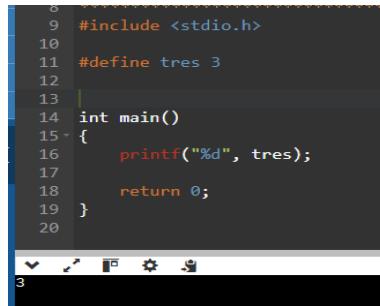


2.3.4 Contantes

Assim como variaveis, constantes também armazenam um valor na memória do computador. A principal diferença para as variaveis é que esse valor não será alterado. Outra coisa é que para as constantes é obrigatoria a atribuição de valor, diferente das variaveis que podemos simplesmente declará-las sem dar um valor

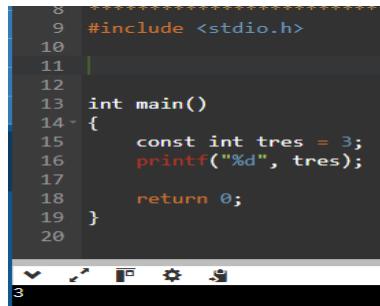
Declaração de constantes

Para declarar uma constante existem duas formas. Na primeira, devemos utilizar `#define nome-constante <valor>` no começo do programa, ANTES DO INICIO DO CÓDIGO. Uma detalhe é que neste caso, não usaremos ponto e vírgula no final. Veja:



```
8
9  #include <stdio.h>
10
11 #define tres 3
12
13
14 int main()
15 {
16     printf("%d", tres);
17
18     return 0;
19 }
20
```

Outra forma é fazer: `const <tipo> nome = valor;`. Como você pôde ver, nesse caso, temos que usar o ponto e vírgula e não precisa estar antes do inicio do código. Veja:



```
8 *****
9  #include <stdio.h>
10
11
12
13 int main()
14 {
15     const int tres = 3;
16     printf("%d", tres);
17
18     return 0;
19 }
20
```

Curiosidade sobre constantes

Já chegamos a falar sobre caracteres de escape (no caso, falamos apenas do barra). Os caracteres de escape são constantes pre-definidas. Veja cada um deles:

Código	Comando
\a	som de alerta (bip)
\b	retrocesso (backspace)
\n	nova linha (new line)
\r	retorno de carro (carriage return)
\v	tabulação vertical
\t	tabulação horizontal
\'	apóstrofe
\"	aspas
\\\	barra invertida (backslash)
\f	alimentação de folha (form feed)
\?	símbolo de interrogação
\0	caractere nulo (cancela a escrita do restante)

2.3.5 Operadores

Os operadores são usados para desenvolver diferentes tipos de operações. Com eles podemos fazer operações matemáticas, comparativas, lógicas e etc. Veremos acerca de cada um dos operadores a seguir

Operadores aritméticos

Os operadores aritméticos são aqueles que operam sobre números e/ou sobre expressões que têm como resultado valores numéricos. Veja os operadores:

Operador	Significado	Exemplo
+	Adição de dois valores	$z = x + y$
-	Subtração de dois valores	$z = x - y$
*	Multiplicação de dois valores	$z = x * y$
/	Quociente de dois valores	$z = x / y$
%	Resto de uma divisão	$z = x \% y$

Um detalhe é que as operações seguem a mesma ordem da matemática. A prioridade são as operações de multiplicação e divisão em detrimento das de soma e subtração.

Outro detalhe é que na divisão, se o numerador e denominador forem inteiros, o compilador retornará apenas a parte inteira da divisão

Operadores relacionais

São aqueles que verificam a magnitude (maior/menor) e/ou igualdades entre dois valores e/ou expressões. Esses operadores retornam verdadeiro (1) ou falso (0) (ou seja, um valor booleano). Veja cada um deles:

Operador	Significado	Exemplo
>	Maior do que	X > 5
\geq	Maior ou igual a	X \geq Y
<	Menor do que	X < 5
\leq	Menor ou igual a	X \leq Z
$=$	Igual a	X == 0
\neq	Diferente de	X != Y

Operadores lógicos

Os operadores lógicos nos permitem representar situações lógicas unindo duas mais expressões relacionais simples em uma composta e nos retornam verdadeiro (1) ou falso (0). Veja cada um deles:

Operador	Significado	Exemplo
$\&\&$	Operador E	(x > 0) $\&\&$ (x < 10)
$\ $	Operador OU	(a == 'F') $\ $ (b != 32)
!	Operador NEGAÇÃO	!(x == 10)

Operadores de atribuição simplificada

Muitas vezes em nosso código nós temos que atribuir valores a nossa variável. Uma forma de fazer isso de maneira mais fácil é utilizando os operadores de atribuição simplificada. Com eles, podemos adicionar valores a nossa variável de forma muito mais simples. Veja cada um deles:

Operador	Significado	Exemplo
$+=$	Soma e atribui	x += y igual a x = x + y
$-=$	Subtrai e atribui	x -= y igual a x = x - y
$*=$	Multiplica e atribui	x *= y igual a x = x * y
$/=$	Divide e atribui o quociente	x /= y igual a x = x / y
$\%=$	Divide e atribui o resto	x %= y igual a x = x % y

Operadores de pré e pós incremento

Esses operadores podem ser utilizados sempre que for necessário somar uma unidade (incremento) ou subtrair uma unidade (decremento) a determinado valor. Veja cada um deles:

Operador	Significado	Exemplo	Resultado
$++$	incremento	$++x$ ou $x++$	$x = x + 1$
$--$	decremento	$--x$ ou $x--$	$x = x - 1$

Um detalhe é que como você pode ver na imagem acima, podemos utilizar o operador antes de depois da variável, mas qual a diferença? Veja abaixo:

Operador	Significado	Resultado
<code>++x</code>	pré-incremento	soma +1 à variável x antes de utilizar seu valor
<code>x++</code>	pós-incremento	soma +1 à variável x depois de utilizar seu valor
<code>--x</code>	pré-decremento	subtrai -1 da variável x antes de utilizar seu valor
<code>x--</code>	pós-decremento	subtrai -1 da variável x depois de utilizar seu valor

2.3.6 Coerção de tipos

Lembra quando falamos anteriormente que se dividirmos um numero inteiro por outro inteiro seu resultado sempre será inteiro, desconsiderando assim a parte decimal? Podemos contornar isso utilizando o casting. O casting é aplicado sobre uma expressão aritmética e força o resultado da expressão a ser de um tipo especificado. Veja as diferentes formas de utilizar o casting:

Type casting explícito

Nós faremos a conversão de tipo no resultado da expressão:

```
int a = 5, b = 2;
float resultado = (float)a / b;
printf("%f\n", resultado); // saída: 2.500000
```

Type casting nos operandos

Faremos o casting nos dois operandos da operação para obter o resultado no tipo que queremos

```
double resultado = (double)a / (double)b;
printf("%lf\n", resultado); // saída: 2.500000
```

2.3.7 Condicionais

Certo. Agora falaremos sobre condicionais. Condicional é basicamente uma mudança de fluxo em nosso código. Caso uma determinada expressão atenda determinada condição, nosso código seguirá por um fluxo e caso contrário, seguirá para outro fluxo. Existem diferentes maneiras de se utilizar as condicionais em nosso código. Veremos cada uma delas abaixo.

If-else

A estrutura do if-else é feita da seguinte forma em nosso código:

```

if (condicao){
    sequencia de comandos 1;
}
else{
    sequencia de comandos 2;
}

```

O que acontece acima é que se a condição for satisfeita, ou seja, for verdadeira (tiver valor 1), nosso programa entrará nesse fluxo e executará o código dentro da condição. Caso contrário, ou seja, caso a condição não for satisfeita (for falsa (ter valor 0)), entraremos no fluxo do else.

Um detalhe é que alem do if e do else, podemos ter ainda o else if, onde caso a condição do if não for satisfeita, haverá a condição do else if a ser satisfeita e aí se ela não satisfeita também, iremos para o else. Veja:

```

if (condicao){
    sequencia de comandos 1;
}
else if (condicao){
    sequencia de comandos 2;
}
else{
    sequencia de comandos 3;
}

```

Swicth-case

O switch-case é outra estrutura de controle de fluxo de código. Sua estrutura é a seguinte:

```

switch (expressao){
    case valor 1:
        sequencia de comandos 1;
        break;

    case valor k:
        sequencia de comandos k;
        break;
    ...
    default:
        sequencia de comandos padrao;
        break;
}

```

O o switch, como podemos ver acima é próprio para testar uma variável em relação a diversos valores pré-estabelecidos. Além disso, como podemos ver acima o default irá desempenhar o valor que o else desempenha na estrutura if-else. Veja um exemplo:

```
ch = getchar();
switch( ch ) {
    case '.':
        printf("Ponto.\n"); break;
    case ',':
        printf("Virgula.\n"); break;
    case ':':
        printf("Dois pontos.\n"); break;
    case ';':
        printf("Ponto e virgula.\n"); break;
    default :
        printf("Nao eh pontuacao.\n");
}
```

2.3.8 Loops

Agora falando sobre loops, o nome já entrega. Os loops serão responsáveis por repetir um bloco de código a partir de uma condição. Enquanto a condição for verdadeira, o bloco de código permanecerá se repetindo. Existem diferentes tipos de loops. Vamos a cada um deles.

While

Significa basicamente "enquanto". Enquanto a condição for verdadeira, ele executará o loop. Veja como é a sintaxe:

```
while( condicao){
    sequencia de comandos;
}
```

OBS: É importante tomar cuidado para não criar uma condição que seja sempre verdadeira, pois isso geraria um loop infinito

Do-While

Esse loop é interessante, pois ele irá executar um bloco de código ao menos uma vez e se a condição no while for verdadeira, ele seguirá executando o bloco de código. A diferença do "do-while" para o "while" é que se a condição já for falsa de inicio, o "while" nem será executado, diferentemente do "do-while", que executa pelo menos uma vez, mesmo com a condição sendo inicialmente falsa. Veja a sintaxe:

```

do{
    sequencia de comandos;
}
while(condicao)

```

Obviamente que assim como o loop while, se a condição for sempre verdadeira, ele executará para sempre

For

O loop for é usado para repetir um número já controlado de vezes. Veja sua sintaxe:

```

for ( inicializcao; condicao; incremento){
    sequencia de comandos;
}

```

Perceba que temos alguns comandos que passamos para o for, sendo eles:

- Inicializacao: iniciar variáveis (contador).
- Condicao: avalia a condição. Se verdadeiro, executa comandos do bloco, senão encerra laço.
- Incremento: ao término do bloco de comandos, incrementa o valor do contado

Perceba então que o loop for executa um número controlado de vezes, a partir do seu contador que passamos na sua inicialização

2.3.9 Arrays

Quando vimos sobre variaveis, estudamos que elas podem armazenar um valor. Sempre que tentamos armazenar um novo valor dentro da variavel o antigo valor é sobrescrito (e portanto, perdido). Agora, pense: E se quisesssemos armazenar mais de um valor em uma variavel? Para isso, usamos os arrays, que é basicamente uma sequencia de elementos do mesmo tipo, onde cada elemento é identificado por um indice. Ou seja, quando criamos um array, nós alocamos um espaço na memória (onde, quanto maior o tamanho do array, que é a quantidade de elementos que ele pode armazenar, maior o espaço de memoria alocado) e podemos armazenar dentro dele varios valores do mesmo tipo (os valores podem ser acessados por meio do indice do elemento dentro do array, que é basicamente a posição do elemento lá dentro). Um exemplo que pode fazer você entender melhor é: suponhamos que queremos armazenar em um local a nota de 5 alunos. Para isso, poderíamos usar um array.

Declaração de arrays

Para declarar um array, nós fazemos da seguinte forma:

<tipo-array> nome-array[tamanho];

Ou seja, primeiro nós precisamos declarar o tipo do array, que será o tipo dos valores que aquele array irá armazenar, depois passamos o nome do array e depois passamos o tamanho do array, ou seja, a quantidade de elementos que ele poderá armazenar.

Inserindo e acessando valores dentro de arrays

Com o array declarado, caso quisermos inserir algum valor no array, basta fazer:

nome-array[indice] = valor;

Lembrando que o indice é a posição do elemento dentro do array. Se tivéssemos um array de tamanho 10 e quiséssemos inserir um valor no sexto elemento, faríamos: nome-array[5] = valor; (uma vez que os indices começariam do 0 e iriam até o 9).

Para acessar valores de um array, basta fazer:

nome-array[indice];

Observações

Em C e C++, se tivermos um array que pode armazenar 10 elementos e tentarmos armazenar 11 elementos, o elemento que sobrar irá ser armazenado em um espaço da memória que não pertence ao array, o que causa comportamento indefinido (pode sobreescrivar dados, travar o programa e entre outros)

Outro detalhe é que se tivermos um array de 10 elementos (ou seja, teremos 10 indices, do 0 ao 9) e tentarmos acessar o indice 10 (11º elemento) o que acontecerá (em C e C++) é que iremos acessar um elemento da memória que não pertence oficialmente ao array, o que pode retornar "Lixos de memória".

2.3.10 Struct - Criação de tipos

Agora, falaremos sobre structs, estamos falando de uma composição de variáveis de outros tipos que formam um "novo tipo de dado". Assim como temos o tipo inteiro, float e etc, podemos "criar" um outro tipo que será um agrupamento de dados.

Declaração

Inserindo valores e acessando valores

2.3.11 Comando typedef

O comando `typedef` nos permite "dar um alias" para os tipos de dados existentes na linguagem C. Se temos, por exemplo, o tipo `float`, mas queremos que ele se chame `flutuante`, poderíamos fazer isso.

Como usar

Para usar, basta fazer o seguinte:

```
typedef <tipo-de-dado> <alias>;
```

O comando acima "da um alias" a um tipo de dado existente. Um detalhe é que o comando acima deve estar no topo do programa, juntamente com a inclusão das bibliotecas.

Exemplo

Capítulo 3

Acerca de ponteiros

3.1 O porquê de estudar esse topico

Agora, iniciaremos um tópico mais avançado, que são os ponteiros. É muito importante entendermos sobre esse conceito porque várias das estruturas de dados que aprenderemos nesta disciplina dependem deles (listas, pilhas, filas, árvores e grafos), então sem entender isso, não iremos para frente

3.2 O que são e como usá-los

3.2.1 O que é

Conceitualmente, um ponteiro é uma variável que armazena o endereço de memória de outra variável. Ou seja, diferentemente das variáveis comuns, um ponteiro não irá armazenar um valor como um caractere, por exemplo, mas sim, um endereço de memória.

3.2.2 Declaração de ponteiros

Para criar um ponteiro, a estrutura lembra bastante a forma como nós criamos as variáveis, mas com pequenas mudanças. Veja como declaramos um ponteiro:

```
<tipo de dado> *nome ponteiro;
```

Perceba que para declarar um ponteiro, assim como nas variáveis, nós temos que usar um tipo de dado. Isso acontece porque nós estamos indicando para o ponteiro que estamos criando o tipo de dado do lugar da memória que ele vai apontar. Isso é importante porque não é muito aconcelhável você ter um ponteiro inteiro e apontar para um char, por exemplo.

Um detalhe é que podemos criar o nosso ponteiro apontando ele para NULL, para que ele não aponte para nenhum lugar (para que consigamos administrar para onde ele aponta depois). Fazendo isso, a declaração ficaria:

```
<tipo de dado> *nome ponteiro = NULL;
```

Se não declararmos da maneira acima e utilizarmos a primeira versão de declaração (`<tipo de dado> *nome ponteiro;`), o que acontece é que nosso ponteiro irá apontar para um endereço de memória aleatório.

Um outro detalhe bem interessante é que podemos fazer com que nosso ponteiro aponte para o endereço de memória de uma variável já existente. Veja:

```
<tipo de dado> *nome ponteiro = &a;
```

Ou seja, o endereço de memória que nosso ponteiro irá apontar, será o endereço da variável `a` (esse significa que estamos nos referindo ao endereço de memoria da variável `a`. Desta forma, o ponteiro `b` apontará para o endereço de memoria de `a`).

3.2.3 Detalhe após a declaração

Quando declaramos um ponteiro(exemplo: `int *a`), algo importante de se dizer é que se utilizarmos `*a` em qualquer outro trecho do nosso código, nós não vamos estar usando o ponteiro em si, mas sim o valor que está no endereço apontado pelo ponteiro `a`. Veja um exemplo:

```
#include <stdio.h>

int main() {
    // 1. Variável de valor
    int x = 18;

    // 2. Declaração do Ponteiro
    // O (*) aqui INDICA que 'ptr' é um ponteiro para um inteiro.
    int *ptr;

    // Ponteiro 'ptr' recebe o ENDEREÇO de 'x' (&x)
    ptr = &x;

    printf("Valor de x antes da mudança: %d\n", x); // Saída: 18

    // 3. Uso do Ponteiro (Desreferência)
    // O (*) aqui ACESSA o valor no endereço apontado por 'ptr' (que é o local
    // e o modifica para 25.
    *ptr = 25;

    printf("Valor de x DEPOIS da mudança: %d\n", x); // Saída: 25

    return 0;
}
```

3.2.4 Exemplo de uso

Veja abaixo um exemplo de uso de ponteiros:

Acima, o que acontece é que:

```
main.c
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 5;
6     int *b = &a;
7
8     printf("a = %d | b = %d\n", a, *b);
9
10    return 0;
11 }
```

- int a = 5; -> Declara uma variável do tipo inteiro chamada a e a inicializa com o valor 5. A variável a armazena o valor 5 em um local específico da memória
- int *b = &a; -> O *b declara uma variável chamada b como um ponteiro para um inteiro (int *). O operador endereço de (&) é usado para obter o endereço de memória onde a variável a está armazenada. O ponteiro b é, portanto, inicializado para armazenar o endereço de memória de a. O valor de b não é 5, mas sim o endereço onde o 5 está guardado.
- parte do printf -> O primeiro %d exibe o valor da variável a, que é 5. O segundo %d exibe o valor armazenado no endereço apontado por b. O operador desreferência ou conteúdo de (*) é usado em frente ao ponteiro b para acessar o valor guardado naquele endereço—ou seja, o valor de a, que também é 5.

Capítulo 4

Falando de funções

4.1 O que é

Funções são blocos de código que podem ser nomeados e chamados dentro de um programa. Elas facilitam a estruturação e reutilização do código, pois:

- Estruturação: programas grandes e complexos são construídos bloco a bloco.
- Reutilização: o uso de funções evita a cópia desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros

4.2 Estrutura

A forma geral de uma função é:



As funções devem ser declaradas ANTES de serem utilizadas, ou seja, antes da cláusula main

4.2.1 Corpo

O corpo é a alma da função e é composto pelos comandos que a função deve executar. Ele processa os parâmetros (se houver), realiza tarefas e gera saídas (se necessário)

Um detalhe é que nós evitamos operações de leitura e escrita dentro de uma função. Essas operações devem ser feitas em quem chamou a função (o main()), por exemplo.

4.2.2 Parâmetros

A declaração de parâmetros é uma lista de variáveis juntamente com seus tipos: tipo1 nome1, tipo2 nome2...

```
//Declaração CORRETA de parâmetros
int soma(int x, int y) {
    return x + y;
}
```

É por meio dos parâmetros que uma função recebe informação do programa principal (isto é, de quem a chamou)

Um detalhe é que podemos criar funções que não recebem nenhum parâmetro. Isso pode ser feito de duas formas:

```
void imprime() {
    printf("Teste\n");
}

void imprime(void) {
    printf("Teste\n");
}
```

4.2.3 Retorno

As funções podem ou não retornar algum valor. Se ela retornar, alguém deverá receber este valor (os valores retornados de funções devem ser armazenados em uma variável)

O retorno da função é dado pelo comando:

```
return valor ou expressao;
```

É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função

Uma função que retorna nada é definida colocando-se o tipo void como valor retornado

```
void imprime(){
    printf("Teste\n");
}
```

Uma função pode ter mais de uma declaração return. Quando um comando return é executado, a função termina imediatamente.

4.2.4 Escopo

Assim como em nosso programa principal, as funções também estão sujeitas ao escopo das variáveis. Escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa:

- Variáveis Locais: São aquelas que só têm validade no bloco onde são declaradas. Exemplo: variáveis declaradas dentro da função
- Variáveis Globais: São declaradas fora de todas as funções do programa. Elas são conhecidas e podem ser alteradas por todas as funções do programa

4.3 Passagem de parâmetros

4.3.1 Passagem por valor

Na linguagem C, os parâmetros de uma função são, por padrão, passados por valor, ou seja, uma cópia do valor do parâmetro é feita e passada para a função. Mesmo que esse valor mude dentro da função, nada acontece com o valor de fora da função

4.3.2 Passagem por referência

Quando se quer que o valor da variável mude dentro da função, usa-se passagem de parâmetros por referência.

Nesse tipo de chamada, não se passa para a função o valor da variável, mas sim sua referência (seu endereço de memória), assim qualquer alteração que a variável sofra dentro da função será refletida fora da função. Um exemplo disso é a função scanf(), onde sempre que desejamos ler um valor, passamos essa função o endereço de memória da variável que queremos armazenar o valor

recolhido, ou seja, a variável que usamos terá seu valor modificado dentro da função scanf(), e seu valor pode ser acessado no programa principal. Veja um outro exemplo:

Como podemos ver acima, para passar um parâmetro por referência, coloca-se um asterisco na frente do nome do parâmetro na declaração da função. A partir disso, ao chamar a função, será necessário passar o operador &, assim como é feito com scanf()

```
//passagem de parâmetro por valor
int x = 10;
incrementa(x);

//passagem de parâmetro por referência
void incrementa(int *n);
//passagem de parâmetro por referência
int x = 10;
incrementa(&x);
```

4.3.3 Arrays como parâmetro

Para utilizar arrays como parâmetros de funções alguns cuidados simples são necessários

- Arrays são sempre passados por referência para uma função. A passagem de arrays por referência evita a cópia desnecessária de grandes quantidades de dados para outras áreas de memória durante a chamada da função, o que afetaria o desempenho do programa
- É necessário declarar um segundo parâmetro (em geral uma variável inteira) para passar para a função o tamanho do array separadamente
- Quando passamos um array por parâmetro, independente do seu tipo, o que é de fato passado é o endereço do primeiro elemento do array

Na passagem de um array como parâmetro de uma função podemos declarar a função de diferentes maneiras, todas equivalentes:

```
void imprime(int *m, int n);
void imprime(int m[], int n);
void imprime(int m[5], int n);
```

4.3.4 Struct como parâmetro

4.4 Recursão

Capítulo 5

Introdução aos algoritmos

5.1 Algoritmos de busca

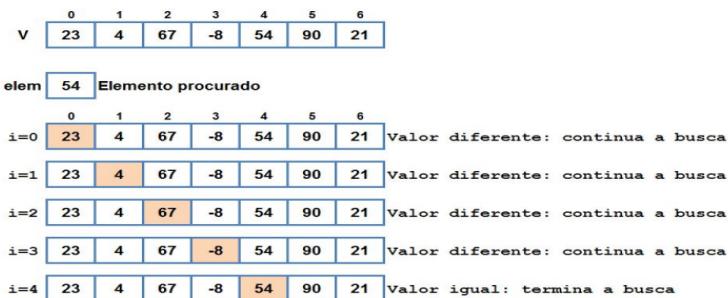
Quando temos um conjunto de dados, podemos querer procurar por um elemento. Exemplo: Temos um array de números inteiros, pode ser que queiramos buscar algum valor dentro desse array.

Existem varios tipos de busca e a utilização dos tipos dependerá de como são os dados (se eles estão estruturados, ordenados e se existem valores duplicados). Com isso em mente, veremos cada um desses tipos de busca

5.1.1 Busca linear - Não ordenada

Como funciona?

Esse é o algoritmo de busca mais simples que existe. O que ele faz é percorrer o array que contém os dados desde sua primeira posição até a última comparando cada valor dele com o valor buscado. Se os valores forem iguais, a busca termina e caso contrário, continua até o fim do array.



Apesar de ser intuitivo, o motivo pelo qual ele tem que percorrer o array inteiro é o fato dele não estar ordenado. No nosso exemplo, suponhamos que o array estivesse ordenado, mas que o número 54 que está sendo buscado não

existisse. Ao fazer nossa busca, quando chegarmos no número 67, ele iria parar a busca, pois o valor procurado não poderia estar depois de 67 (o array está ordenado).

Complexidade

A complexidade do algoritmo de busca linear não ordenada pode ser analisada conforme os seguintes casos:

- Melhor caso: O(1). Acontece quando o elemento buscado é o primeiro do array.
- Pior caso: O(N). Acontece quando o elemento é o último do array ou não existe.
- Caso médio: O(N/2).

Implementação

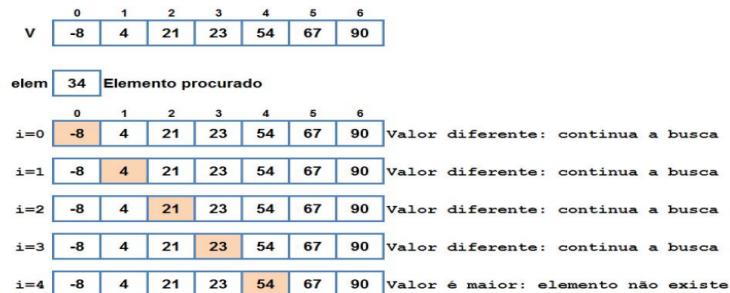
```

13
14 int buscaLinear(int *V, int N, int elem) {
15     int i;
16     for(i = 0; i < N; i++) {
17         if(elem == V[i])
18             return i; //elemento encontrado
19     }
20     return -1; //elemento não encontrado
21 }
```

5.1.2 Busca linear - Ordenada

Como funciona?

A busca sequencial ordenada funcionará da mesma forma que a busca sequencial não ordenada. A diferença é que, com o array ordenado, caso o valor do array seja maior que o valor buscado, ele parará a busca.



Complexidade

A complexidade do algoritmo de busca linear ordenada pode ser analisada conforme os seguintes casos:

- Pior caso: $O(n)$. Acontece quando o valor é o maior valor do array, ou seja, está na última posição do array

Implementação

```
14
15 int buscaOrdenada(int *V, int N, int elem) {
16     int i;
17     for(i = 0; i < N; i++) {
18         if(elem == V[i])
19             return i;//elemento encontrado
20         else
21             if(elem < V[i])
22                 return -1;//para a busca
23     }
24     return -1;//elemento não encontrado
25 }
```

5.1.3 Busca Binária

Como funciona?

Esse algoritmo é uma das formas mais "especializadas" de se realizar uma busca em um array. Para utilizá-lo o array DEVE estar ordenado. O que ele faz é calcular o meio do array e utilizar o valor desse meio para comparar com o valor buscado. Se o valor buscado for menor que o valor do meio, ele descarta a segunda metade do array e fica com apenas a primeira metade com os valores menores. Caso o valor buscado for maior, ele fará o contrário



A parte em azul na imagem acima representa a parte do array que foi descartada por conta da condição

Um detalhe é que ao realizar a comparação entre o valor buscado e o valor do meio do array, ele vai verificar se o valor buscado é igual ao valor do meio do array e, caso for, ele já encerra a busca ali mesmo.

Complexidade

A complexidade do algoritmo de busca linear ordenada pode ser analisada conforme os seguintes casos:

- Melhor caso: O(1). O elemento está exatamente no meio do array
- Caso médio: O(log2 N).
- Pior caso: O(N). O elemento não existe

Implementação

Abaixo, a sua implementação no código:

```
27 | 
28 | int buscaBinaria(int *V, int N, int elem) {
29 |     int i, inicio, meio, final;
30 |     inicio = 0;
31 |     final = N-1;
32 |     while(inicio <= final) {
33 |         meio = (inicio + final)/2;
34 |         if(elem < V[meio])
35 |             final = meio-1;//busca na metade da esquerda
36 |         else
37 |             if(elem > V[meio])
38 |                 inicio = meio+1;//busca na metade da direita
39 |             else
40 |                 return meio;
41 |     }
42 |     return -1;//elemento não encontrado
43 | }
```

5.1.4 Busca em array de struct

Como funciona?

Aqui, estamos lidando com algo mais complexo. Quando queremos buscar algo em um array de struct, nós usaremos uma das chaves da struct para realizar a busca, como usar a matrícula para buscar de uma struct para realizar uma busca

```
struct aluno V[6];
```

matricula;	matricula;	matricula;	matricula;	matricula;	matricula;
nome[30];	nome[30];	nome[30];	nome[30];	nome[30];	nome[30];
n1,n2,n3;	n1,n2,n3;	n1,n2,n3;	n1,n2,n3;	n1,n2,n3;	n1,n2,n3;

V[0] v[1] v[2] v[3] v[4] v[5]

Complexidade

Como estamos falando de uma busca linear, sua complexidade será a mesma

Implementação

```
28 int buscaLinearMatricula(struct aluno *V, int N, int elem){  
29     int i;  
30     for(i = 0; i < N; i++){  
31         if(elem == V[i].matricula)  
32             return i;//elemento encontrado  
33     }  
34     return -1;//elemento não encontrado  
35 }
```

5.2 Algoritmos de ordenação

Após uma base de dados estar construída pode ser necessário ordená-la. A ordenação dos dados PODE ser um passo preliminar para pesquisá-los (para utilizar o algoritmo de busca binária, por exemplo, precisamos que os dados estejam ordenados). Dada essa introdução, veremos alguns algoritmos de ordenação de dados.

5.2.1 Bubblesort

Como funciona?

O algoritmo de ordenação bubblesort é o mais simples dos algoritmos de ordenação. O que ele faz é:

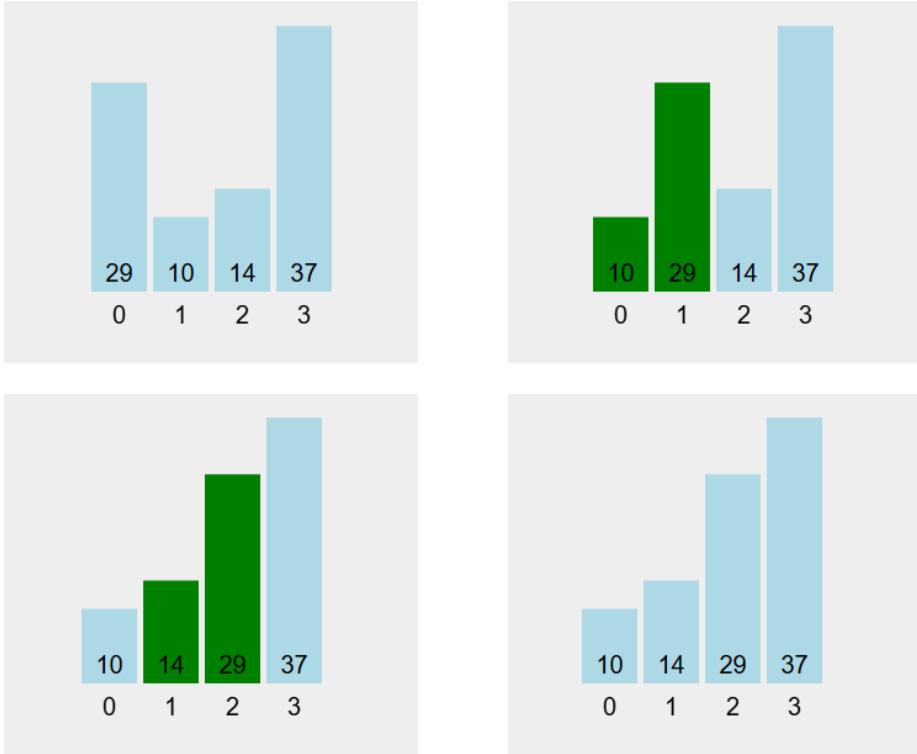
1. Comparação de dois números
2. Se o da esquerda for maior, os elementos devem ser trocados
3. Desloca-se uma posição à direita

Exemplo: Estamos percorrendo um vetor e na posição da esquerda nós temos o número 10 e na posição da direita nós temos o número 8. Como 8 é menor que 10, iremos fazer a troca. No lugar do 10, teremos o 8 e no lugar do 8 teremos o 10. Feito isso, ele estará na posição onde está o valor 8, então ele irá se deslocar uma posição à direita (que é onde estará o valor 10).

A medida que o algoritmo avança, os itens maiores "surgem como uma bolha" na extremidade superior do vetor (à direita do vetor). É por isso que o é o algoritmo da bolha (bolha = bubble)

Podemos observar essa algoritmo de forma visual a partir deste link. Aqui vai um exemplo rápido com algumas imagens:

Na primeira imagem, podemos ver o nosso vetor de forma desordenada, aí o que acontecerá com a aplicação do algoritmo de bubble sort é que pegaremos o número da esquerda (no caso 29) e iremos comparar com o segundo. Se o segundo número for menor, jogamos o maior para a direita (assim como podemos ver na segunda imagem, onde o número 29 foi para a direita e o 10, que era menor, foi para a esquerda). Após isso, faremos a comparação novamente entre o

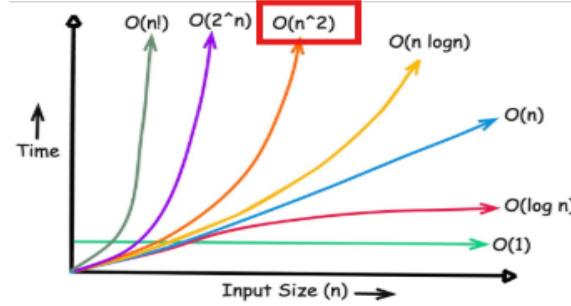


número da esquerda e o da direita. Agora, estamos comparando o 29 (esquerda) e o 14 (direita), então como o número da esquerda é maior (29), iremos jogá-lo para a direita e o número que estava na direita virá para a esquerda, assim como podemos ver na terceira imagem. Por fim, vamos fazer a comparação novamente entre o número da esquerda e o da direita. Dessa vez o número da esquerda é menor que o da direita, então não haverá troca, aí passaremos para o próximo número (37) para realizar novas comparações, mas como o vetor acabou, então finalizamos por aqui.

Portanto, o algoritmo vai percorrendo o vetor e fazendo as trocas, mas pode ser que ele tenha que percorrer o vetor mais de uma vez para fazer a ordenação (na maioria das vezes é o que acontece, apesar de termos dado sorte no exemplo que demos acima). Apesar disso, uma coisa que ele garante é que após a primeira rodada o maior elemento do array será movido para a última posição do array e isso faz com que, para um vetor com n elementos, o Bubble Sort precise de no máximo $n-1$ passagens. Isso acontece porque a cada passagem, o maior elemento restante é movido para sua posição final correta. Após a $n-1^{\text{a}}$ passagem, os $n-1$ maiores elementos já estarão ordenados. Por consequência, o único elemento que sobrou, o menor de todos, já estará automaticamente na primeira posição, que é a sua posição correta. Sendo assim, não há necessidade de uma n -ésima passagem.

Complexidade

Com relação ao Big-O desse algoritmo, ele é um $O(n^2)$, ou seja, o tempo de execução dele é relativamente grande:



A razão que faz esse algoritmo ser um $O(n^2)$ é o fato de ter dois loops aninhados que o algoritmo usa para percorrer o vetor (veremos na implementação)

Implementação

```
#include <stdio.h>

//Funcao para trocar dois elementos de lugar
void trocar(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

//Funcao que implementa o Bubble Sort
void bubbleSort(int vetor[], int n) {
    int i, j;
    int houveTroca;

    //O algoritmo precisa repetir varias vezes
    //ate que nao haja mais trocas
    for (i = 0; i < n - 1; i++) {
        houveTroca = 0; //no começo da rodada, não houve troca ainda

        //Percorre o vetor ate a penultima posição comparando os vizinhos
        for (j = 0; j < n - i - 1; j++) {
            if (vetor[j] > vetor[j + 1]) {
                trocar(&vetor[j], &vetor[j + 1]);
                houveTroca = 1; // se houve troca, marcamos
            }
        }
    }
}
```

```

        //Se nao houve troca , significa que o vetor ja esta ordenado
        if (houveTroca == 0) {
            break;
        }
    }

//Funcao para imprimir o vetor
void imprimirVetor(int vetor[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
}

int main() {
    int vetor[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(vetor) / sizeof(vetor[0]);

    printf("Vetor original: ");
    imprimirVetor(vetor, n);

    bubbleSort(vetor, n);

    printf("Vetor ordenado: ");
    imprimirVetor(vetor, n);

    return 0;
}

```

Explicação da implementação:

1. void trocar(int *a, int*b)

Essa é uma função auxiliar que serve para trocar os valores de duas variáveis inteiras.

- int *a e int *b: Os parâmetros são ponteiros para inteiros, e não as variáveis em si. Como sabemos, um ponteiro é uma variável que armazena o endereço de memória de outra variável. Usamos ponteiros aqui porque, para a função trocar realmente modificar os valores das variáveis que a chamaram, ela precisa de seus endereços de memória. Se passássemos apenas int a e int b, a função criaria cópias locais dos valores, e a troca não afetaria o vetor original.
- int temp = *a;: A variável temp (de "temporário") é usada para guardar o valor original do primeiro elemento. O asterisco (*) é

o operador de desreferenciação; ele "desempacota" o ponteiro para acessar o valor que está no endereço de memória. Então, `*a` é o valor da variável que o ponteiro a está apontando.

- `*a = *b;`: O valor da segunda variável (`*b`) é atribuído à primeira (`*a`)
- `*b = temp;`: O valor original da primeira variável (`*a`), que estava guardado em `temp`, é atribuído à segunda (`*b`).

2. void bubbleSort(int vetor[], int n)

Esta é a função principal que implementa o algoritmo de ordenação.

- `for (i = 0; i < n - 1; i++)`: Este é o laço externo. Ele controla o número de "passagens" que o algoritmo fará pelo vetor. Em cada passagem, o maior elemento "flutua" para a sua posição correta no final do vetor. O `n - 1` é usado porque, se temos `n` elementos, precisamos de no máximo `n - 1` passagens para ordená-los.
- `int houveTroca = 0;`: Esta variável de controle é uma otimização do Bubble Sort. Ela é inicializada como 0 (falso) no início de cada passagem. Se o laço interno não realizar nenhuma troca, significa que o vetor já está ordenado, e podemos parar o algoritmo mais cedo.
- `for (j = 0; j < n - i - 1; j++)`: Este é o laço interno. Ele é responsável por percorrer o vetor e comparar os pares de elementos vizinhos.
 - `n - i - 1`: A cada passagem do laço externo (`i`), o maior elemento já está na sua posição correta no final. Portanto, não precisamos mais comparar os elementos que já estão no lugar certo. Por exemplo, na primeira passagem (`i=0`), o maior elemento vai para a última posição. Na segunda passagem (`i=1`), o segundo maior elemento vai para a penúltima posição, e assim por diante. Essa otimização evita comparações desnecessárias, melhorando a eficiência do algoritmo.
- `if (vetor[j] > vetor[j + 1])`: Esta é a condição principal de comparação. Se o elemento atual (`vetor[j]`) for maior que o seu vizinho da direita (`vetor[j + 1]`), eles estão na ordem errada para uma ordenação crescente.
- `trocar(&vetor[j], &vetor[j + 1]);`: Com a condição sendo verdadeira, a função `trocar` é chamada. Note que usamos o operador `&` (operador de endereço) para passar o endereço de memória dos elementos do vetor, pois a função `trocar` espera ponteiros.
- `if (houveTroca == 0) break;`: Esta é a otimização comentada antes. Se, depois de uma passagem completa do laço interno, a variável `houveTroca` ainda for 0, significa que o vetor está totalmente ordenado. Nesse caso, usamos o comando `break` para sair do laço externo, encerrando o algoritmo.

3. int main()

Esta é a função principal do programa, onde a execução começa.

- `int vetor[] = 64, 34, 25, 12, 22, 11, 90;;` Declara e inicializa um vetor de inteiros com os valores a serem ordenados.
- `int n = sizeof(vetor) / sizeof(vetor[0]);;` Esta é uma forma padrão e portátil de calcular o número de elementos em um vetor em C.
 - `sizeof(vetor)`: Retorna o tamanho total do vetor em bytes.
 - `sizeof(vetor[0])`: Retorna o tamanho de um único elemento do vetor em bytes (neste caso, o tamanho de um int).
 - Ao dividir o tamanho total pelo tamanho de um elemento, obtemos o número exato de elementos no vetor, independentemente do tipo de dado ou da arquitetura do sistema. Isso é muito mais robusto do que simplesmente contar os elementos manualmente.
- `printf("Vetor original: "); e imprimirVetor(vetor, n);;` Exibe o vetor antes da ordenação.
- `bubbleSort(vetor, n);;` Chama a função para ordenar o vetor.
- `printf("Vetor ordenado: "); e imprimirVetor(vetor, n);;` Exibe o vetor após a ordenação.

5.2.2 Insertion sort

Como funciona?

Também é um dos mais simples algoritmos de ordenação existentes. Ele possui um método de ordenação semelhante ao que usamos para ordenar as cartas de um baralho. O que ele faz é pegar uma carta de cada vez e a coloca em seu devido lugar, sempre deixando as cartas da mão em ordem.

1.

Complexidade

Os casos de complexidade

- Melhor caso: $O(n)$. Ocorre quando a lista já está completamente ordenada.
- Caso médio: $O(n^2)$.
- Pior caso: $O(n^2)$.

5.2.3 Selection sort

Como funciona?

A idéia da ordenação por seleção é procurar o menor elemento do vetor (ou maior) e movimentá-lo para a primeira (última) posição do vetor. Repetir para os n elementos do vetor.

Complexidade

Sua complexidade:

- Melhor caso: $O(n^2)$
- Caso médio: $O(n^2)$
- Pior caso: $O(n^2)$

5.2.4 Merge sort

Como funciona?

Também conhecido como ordenação por intercalação. É um Algoritmo recursivo que usa a idéia de dividir para conquistar para ordenar os dados (Parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um com muitos). O algoritmo divide os dados em conjuntos cada vez menores para depois ordená-los e combina-los por meio de intercalação (merge)

- Divide, recursivamente, o conjunto de dados até que cada subconjunto possua 1 elemento
- Combina 2 subconjuntos de forma a obter 1 conjunto maior e ordenado
- Esse processo se repete até que exista apenas 1 conjunto

Complexidade

- Melhor caso: $O(n \log n)$
- Caso médio: $O(n \log n)$
- Pior caso: $O(n \log n)$

5.2.5 Shell sort

Como funciona?

Complexidade

5.2.6 Quicksort

Como funciona?

É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações. Provavelmente é o mais utilizado. Ideia básica: Dividir e Conquistar. Um elemento é escolhido como pivô. “Particionar”: os dados são rearranjados (valores menores do que o pivô são colocados antes dele e os maiores, depois). Recursivamente ordena as 2 partições

Complexidade

- Melhor caso: $O(n \log n)$
- Caso médio: $O(n \log n)$
- Pior caso: $O(n^2)$

Capítulo 6

Alocação dinâmica

6.1 Conceito

Sempre que escrevemos um programa, é preciso reservar espaço para as informações que serão processadas. Para isso, utilizamos as variáveis.

Infelizmente, nem sempre é possível saber, em tempo de execução, o quanto de memória um programa irá precisar.

Com isso, a alocação dinâmica permite ao programador alocar memória em tempo de execução, ou seja, a quantidade de memória é alocada sob demanda, quando o programa precisa, e não apenas quando se está escrevendo o programa.

Veja um exemplo abaixo:

Memória		
posição	variável	conteúdo
119		
120		
121	int *p	NULL
122		
123		
124		
125		
126		
127		
128		

Alocando 5 posições de memória em int *p

Memória		
posição	variável	conteúdo
119		
120		
121	int *p	123
122		
123	p[0]	11
124	p[1]	25
125	p[2]	32
126	p[3]	44
127	p[4]	52
128		

O que acontece acima é que nós temos um ponteiro p que não aponta para nada, aí, usando a alocação dinâmica em tempo de execução no nosso programa, nós alocamos 5 espaços de memória para p. Na prática, é como se estivéssemos criando um array (na verdade é exatamente isso)

6.2 Tipos de alocação dinâmica

6.2.1 malloc

Como funciona?

A função malloc() serve para alocar memória e tem o seguinte protótipo:

```
void *malloc (unsigned int num);
```

Assim, dado o número de bytes que queremos alocar (num), ela aloca na memória e retorna um ponteiro void* para o primeiro byte alocado.

Exemplo

Vamos ter um exemplo onde queremos alocar 1000 bytes de memória livre:

```
char *p;  
p = (char *) malloc(1000);
```

Agora um exemplo vamos alocar espaço de memória para 50 inteiros:

```
int *p;  
p = (int *) malloc(50*sizeof(int));
```

Uma observação importante é que a função sizeof() cálcula o tamanho do objeto que você passa para ele em bytes, então acima nós calculamos o tamanho de um inteiro em bytes e multiplicamos por 50. Além disso, outro detalhe é que se não houver memória suficiente para alocar a memória requisitada, a função malloc() retorna um ponteiro nulo

6.2.2 calloc

Como funciona?

A função calloc() também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
void *calloc (unsigned int num, unsigned int  
             size)
```

Basicamente, a função calloc() faz o mesmo que a função malloc(). A diferença é que agora passamos a quantidade de posições a serem alocadas e o tamanho do tipo de dado alocado como parâmetros distintos da função.

Exemplo

Vamos ver um exemplo no código a seguir:

```
int *p1;  
p1 = (int *) calloc(50, sizeof(int));  
if(p1 == NULL){  
    printf("Erro: Memoria Insuficiente!\n");  
}
```

Perceba acima que diferentemente do malloc, você passa os parâmetros de forma distinta (isso pode ser visto separando eles por vírgula). Além disso, note o que falamos anteriormente que se não houver memória suficiente para alocar, será retornado um ponteiro nulo

6.2.3 realloc

Como funciona?

A função realloc() serve para REALOCAR memória e tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num);
```

A função realloc irá modificar o tamanho da memória previamente alocada e apontada por *ptr para o valor indicado por num (sendo que num pode ser maior ou menor do que a quantidade de memória alocada)

Alguns detalhes é que: Se num for 0, a memória apontada por *ptr é liberada (assim como a função free que veremos em breve). Além disso, se *ptr for nulo, o numero de bytes será alocado e devolverá um ponteiro (assim como o malloc faz)

Exemplo

Vamos ver um exemplo abaixo com o contexto de um malloc no começo do programa de $5 * \text{sizeof}(\text{int})$

```
p = realloc(p, 3*sizeof(int));
for (i = 0; i < 3; i++) {
    printf("%d\n", p[i]);
}
printf("\n");
//Aumenta o tamanho do array
p = realloc(p, 10*sizeof(int));
for (i = 0; i < 10; i++) {
    printf("%d\n", p[i]);
}
```

6.2.4 free

Como funciona?

Quando alocamos memória, nós estamos, na verdade, alocando um array. Ok, isso já sabemos. Para desalocar essa memória basta utilizar a função free() passando como argumento o ponteiro para a memória alocada. Veja abaixo:

Exemplo

Veja um exemplo abaixo da aplicação disso:

```

int *p;
int i, N = 100;

p = (int *) malloc(N*sizeof(int));

for (i = 0; i < N; i++)
    scanf("%d", &p[i]);

free(p);

```

Perceba acima que nós fizemos o alocação e no final fizemos a liberação

6.3 Casos especiais da alocação dinâmica

6.3.1 Arrays

Como funciona?

Quando temos arrays com mais de uma dimensão, utilizamos o conceito de ponteiro para ponteiro. Veja como esse conceito funciona visualmente:

```

char letra = 'a';
char *p1;
char **p2;
char ***p3;

p1 = &letra;
p2 = &p1;
p3 = &p2;

```

Memória		
posição	variável	conteúdo
119		
120	char ***p3	122
121		
122	char **p2	124
123		
124	char *p1	126
125		
126	char letra	'a'
127		

Acima, nós vamos ter a variável que contém o caractere 'a' e ai vamos ter o ponteiro p1 que aponta para o endereço de memória dessa variável, o ponteiro p2 que aponta para o endereço de memória de p1 e p3 que aponta para o endereço de memória de p2. Se mudarmos o endereço de memória que p1 aponta, todo o resto muda.

Entendido isso, veja como funcionará quando queremos alocar memória para um array de mais de uma dimensão (especificamente 2 dimensões):

```

int **p; // 2 "niveis = 2 dimensões
int i, j, N = 2;
p = (int**) malloc(N*sizeof(int*));

for (i = 0; i < N; i++) {
    p[i] = (int *)malloc(N*sizeof(int));
    for (j = 0; j < N; j++)
        scanf("%d", &p[i][j]);
}

```

Memória		
posição	variável	conteúdo
119	int **p	120 ↓
120	p[0]	123
121	p[1]	126
122		
123	p[0][0]	69
124	p[0][1]	74
125		
126	p[1][0]	14
127	p[1][1]	31
128		

No exemplo acima, o que aconteceu foi:

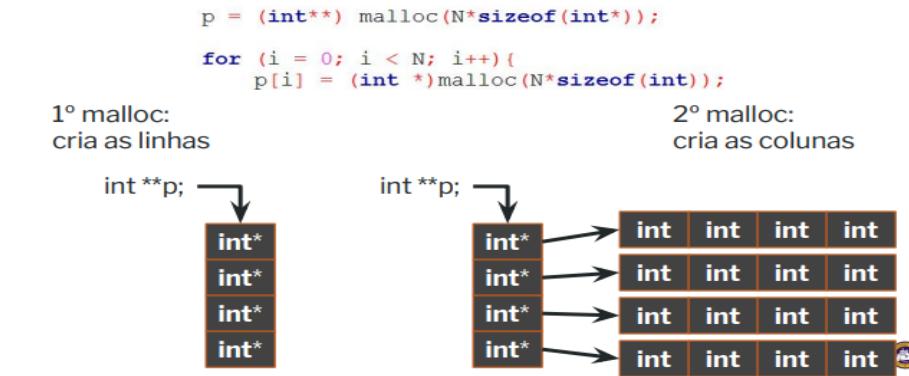
1. int **p;
 - Isso declara p como um "ponteiro para um ponteiro de inteiro".
2. int i, j, N = 2;
 - Aqui, declaramos tres variaveis do tipo inteiro. i e j serão utilizadas como contadores nos loops (para percorrer linhas e colunas). A variavel N é definida com o valor 2, o que significa que
3. p = (int**) malloc(N*sizeof(int*));
 - Esta é a primeira alocação de memória. Utilizando o malloc você está alocando espaço para N ponteiros de inteiro. Assim, p (sendo um ponteiro para ponteiros de inteiro) irá apontar para o bloco de memória p[0] (que é um ponteiro para inteiro) e seguinte ao bloco p[0], temos p[1]. No momento, os ponteiros para inteiros que temos em p[0] e p[1] estão vazios (tecnicamente, apontam para lixo)
4. for (i = 0; i < N; i++) ...
 - Este é o loop externo. Ele vai ser executado N (2) vezes: uma para i = 0 (primeira linha) e outra para i = 1 (segunda linha).
5. p[i] = (int *)malloc(N*sizeof(int));
 - Essa é a segunda alocação de memória. Ela acontece dentro do loop. Nós iremos alocar um espaço na memória para N inteiros.
 - Quando i = 0, por exemplo, para o ponteiro para inteiros que fica em p[0], ele irá apontar para o endereço de memória onde teremos o nosso primeiro valor e inteiro
6. for (j = 0; j < N; j++) ...
 - Este é o loop interno. Ele está aninhado (dentro) do primeiro loop.

- Quando $i = 0$, após $p[i] = (\text{int } *)\text{malloc}(N*\text{sizeof}(\text{int}))$; ter sido executado, $p[0]$ terá um ponteiro que aponta para um endereço de memória que armazena um inteiro e seguido desse endereço, tem um outro endereço de memória que armazena outro inteiro (eles ficam um seguido do outro). Aí que entra o j do loop. Para armazenar os inteiros nesse endereço de memória, utilizaremos o j para percorrer-los

7. `scanf("%d", &p[i][j]);`

- `scanf` é a função que lê dados do teclado
- `"%d"` diz a ela que esperamos um número inteiro.
- `&p[i][j]` é onde a mágica acontece:
 - Em $p[i]$, estamos acessando o o lugar onde temos o ponteiro para o inteiro, então em $p[i][j]$ estamos acessando o local para onde aquele ponteiro para inteiros aponta. Esse local é o que armazenará o valor do inteiro
 - O `&` (operador "endereço de") passa o endereço exato daquela posição na memória para o `scanf`, para que ele saiba onde salvar o número que o usuário digitar.

Veja ainda o processo explicado acima de forma mais visual:



Desalocação

Agora que já vimos sobre a alocação de memória em arrays de mais de uma dimensão, precisamos ver sobre liberação (desalocamento). Na liberação, ocorre de maneira inversa do alocamento. Nós vamos desalocar a memória primeiro dos ponteiros para inteiros e depois do ponteiro para ponteiros de inteiros. Veja:

```

int **p; //2 "*" = 2 níveis = 2 dimensões
int i, j, N = 2;
p = (int**) malloc(N * sizeof(int*));

for (i = 0; i < N; i++) {
    p[i] = (int *) malloc(N * sizeof(int));
    for (j = 0; j < N; j++)
        scanf("%d", &p[i][j]);
}

for (i = 0; i < N; i++)
    free(p[i]);
free(p);

```

6.3.2 Structs

Assim como nos tipos básicos, também é possível realizar a alocação dinâmica de structs.

Como funciona

Podemos realizar a alocação de uma única struct ou de mais de uma:

- Uma única struct
 - Um ponteiro para struct receberá o malloc
 - Utilizamos o operador seta para acessar o conteúdo
 - Usamos o free para liberar a memória alocada

```

struct cadastro{
    char nome[50];
    int idade;
};

int main(){
    struct cadastro *cad = (struct cadastro*) malloc(sizeof(struct cadastro));
    strcpy(cad->nome, "Maria");
    cad->idade = 30;

    free(cad);

    return 0;
}

```

- Mais de uma struct
 - Um ponteiro para a struct receberá o malloc
 - Utilizamos os colchetes para acessar o conteúdo
 - Usamos o free para liberar a memória alocada

```
struct cadastro{
    char nome[50];
    int idade;
};

int main(){
    struct cadastro *cad = (struct cadastro*) malloc(sizeof(struct cadastro));
    strcpy(cad->nome, "Maria");
    cad->idade = 30;

    free(cad);

    return 0;
}
```

Capítulo 7

Listas encadeadas, Pilha e Fila

7.1 Listas encadeadas

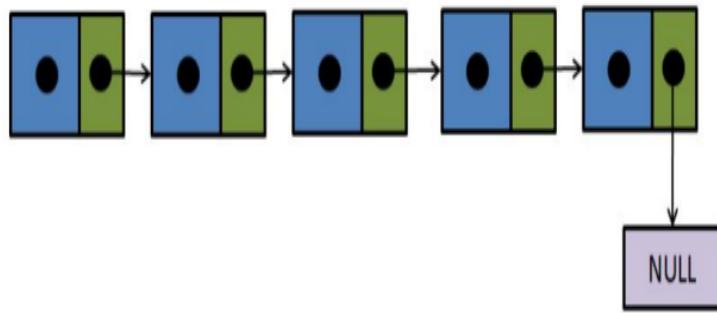
A lista encadeada lembra bastante os arrays, mas diferente deles onde seus elementos são guardados em posições contíguas de memória, uma a lista encadeada é formada por uma sequência de nós (ou nodes) que estão espalhados pela memória. A vantagem é justamente em operações de inserção ou remoção não precisar movimentar os elementos na memória (uma vez que não estamos falando de posições contíguas)

7.1.1 Como funciona?

Cada nó em uma lista encadeada é composto por duas partes principais:

1. Dado: O valor ou informação que se deseja armazenar (um número, um texto, um objeto, etc)
2. Ponteiro: Um endereço de memória que "aponta" para o próximo nó da sequência.

A lista "sabe" onde começa através de um ponteiro especial chamado cabeça (ou head), que aponta para o primeiro nó (é um ponteiro para ponteiro). O último nó da lista tem um ponteiro especial que aponta para NULL (nulo), indicando que a sequência terminou. Veja a imagem abaixo:



7.1.2 Implementação

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

//codigo pra liberar lista de tras pra frente.

typedef struct cel{
    int conteudo;
    struct cel *seg;
} cel;

typedef struct cel* Lista;

void imprime_lista(Lista* lista){
    printf("\nx");
    if(lista==NULL){
        printf("\n1");
        return;
    }
    cel* aux = *lista;
    printf("\n2");
    while(aux!=NULL){
        printf("\t%i ",aux->conteudo);
        aux=aux->seg;
    }
    printf("\n");
}

Lista* cria_lista(){
    Lista *li = (Lista*) malloc(sizeof(Lista));
    li->conteudo = 1;
    li->seg = NULL;
    return li;
}
```

```

        if( li != NULL){
            *li=NULL;
        }
        return li;
    }

int insere_lista_fim(Lista* lista , int x){
    if(lista==NULL) {return 0;}
    cel* aux = (cel*) malloc(sizeof(cel));
    if(aux==NULL){return 0;}
    aux->conteudo = x;
    aux->seg = NULL;
    if((*lista)==NULL){ *lista = aux;}
    else{
        cel *temp;
        temp = *lista ;
        while(temp->seg!= NULL){ //caminha ate o ultimo elemento
            temp= temp->seg;
        }
        temp->seg = aux;
    }
    return 1;
}

int busca(Lista *lista , int valor){
    cel *p;
    for(p=*lista ;p!=NULL;p=p->seg){
        if(p->conteudo == valor){
            return 1;
        }
    }
    return 0;
}

void libera_lista(Lista* lista){
    if( lista!=NULL){
        cel* aux;
        while(*lista!=NULL){
            aux = *lista ;
            *lista =(*lista)->seg;
            free(aux);
        }
        free(lista );
    }
}

```

```

int remove_lista(Lista* lista , int x){
    if (lista==NULL){ return 0;}
    if ((*lista)==NULL){ return 0;}
    cel *ant , *aux=*lista ;
    while (aux!=NULL && aux->conteudo !=x){
        ant=aux ;
        aux=aux->seg ;
    }
    if (aux==NULL){ return 0;}

    if (aux==*lista){
        *lista = aux->seg ;
    }
    else{
        ant->seg = aux->seg ;
    }
    free(aux);
    return 1;
}

int main(void) {
    Lista *lst ;
    lst = cria_lista();
    insere_lista_fim(lst , 1);
    insere_lista_fim(lst , 2);
    insere_lista_fim(lst , 3);
    insere_lista_fim(lst , 4);
    insere_lista_fim(lst , 5);
    imprime_lista(lst );

    remove_lista(lst ,4);
    imprime_lista(lst );
    printf("\nA busca retornou %d\n",busca(lst ,7));
    libera_lista(lst );
    imprime_lista(lst );

    return 0;
}

```

Vamos destrinchar o código acima para entender cada uma de suas partes:

```

typedef struct cel{
    int conteudo;
    struct cel *seg;
} cel;

```

Acima, nós temos uma struct, que representará os nós. Cada nó da lista é uma célula (cel), com dois campos:

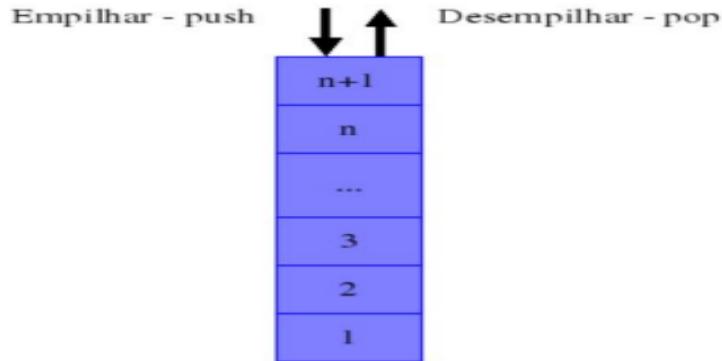
- conteudo: o valor armazenado (no caso, um int);
- seg: ponteiro para o próximo nó da lista.

7.2 Pilha

Pilhas e Filas são como listas, porém elas tem regras estritas sobre como os elementos podem ser adicionados e removidos.

7.2.1 Como funciona

As pilhas são estruturas do tipo LIFO (last-in first-out). Isso significa que o último elemento inserido é o primeiro a sair. Além disso, nós só teremos acesso ao último elemento. Para processar o penúltimo elemento, devemos remover o último



7.2.2 Curiosidade

Uma curiosidade é que tem alguns exemplos interessantes onde a pilha é utilizada, são eles:

- Botão "Desfazer" (Undo): Editores de texto guardam as suas ações (digitar, apagar) numa pilha. Quando você clica em "Desfazer", ele dá pop na última ação e a reverte (a ultima ação que você fez é a primeira revertida).
- Histórico de Navegação: O botão "Voltar" do seu navegador funciona como uma pilha. Cada página que você visita é "empilhada" (push). Ao clicar em "Voltar", ele "desempilha" (pop) a página atual e mostra a anterior. (a pagina mais recente que você acessou é "desempilhada")
- Chamadas de Funções: Quando o seu código chama uma função, o computador "empilha" a localização atual para saber para onde voltar quando a função terminar.

7.2.3 Implementação

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

struct elemento{
    int conteudo;
    struct elemento *prox;
};

typedef struct elemento Elem;
typedef struct elemento* Pilha;

Pilha* cria_pilha(){
    Pilha* pi = (Pilha*) malloc(sizeof(Pilha));
    if(pi!=NULL){
        *pi=NULL;
    }
    return pi;
}

void libera_pilha(Pilha* pi){
    if(pi!=NULL){
        Elem* no;
        while((*pi)!=NULL){
            no = *pi;
            *pi=(*pi)->prox;
            free(no);
        }
        free(pi);
    }
}

int consulta_topo(Pilha* pi){
    if(pi==NULL){
        return 0;
    }
    if((*pi)==NULL){
        return 0;
    }
    return (*pi)->conteudo;
}

int insere_Pilha(Pilha* pi,int x ){
    if(pi==NULL){
        return 0;
```

```

    }
    Elemt* no;
    no = (Elemt*) malloc(sizeof(Elemt));
    if (no==NULL){
        return 0;
    }
    no->conteudo = x;
    no->prox = (*pi);
    *pi=no;
    return 1;
}

int remove_Pilha(Pilha* pi){
    if (pi==NULL){
        return 0;
    }
    if ((*pi)==NULL){
        return 0;
    }
    Elemt *no = *pi;
    *pi = no->prox;
    free(no);

    return 1;
}

void imprime_Pilha(Pilha* pi){
    if (pi==NULL){
        return;
    }
    Elemt* no =*pi;
    printf("\n-----Pilha-----");
    while (no!=NULL){
        printf("\nConteudo: %d\n", no->conteudo);
        no = no->prox;
    }
    printf("\n-----");
}

int main(void) {
    Pilha* pi = cria_pilha();
    insere_Pilha(pi, 0);
    insere_Pilha(pi, 1);
}

```

```

insere_Pilha(pi , 2);
insere_Pilha(pi , 3);
insere_Pilha(pi , 4);
imprime_Pilha(pi );

remove_Pilha(pi );
remove_Pilha(pi );

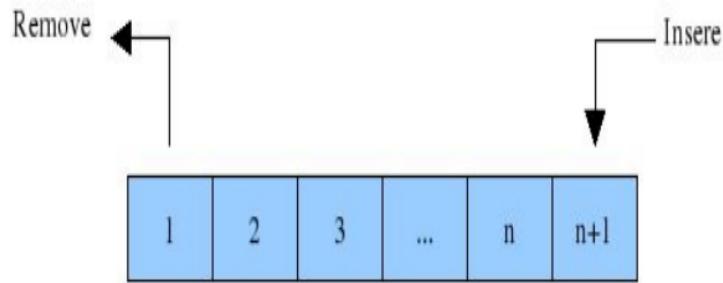
imprime_Pilha(pi );
return 0;
}

```

7.3 Fila

7.3.1 Como funciona

As pilhas são estruturas do tipo FIFO (first-in first-out). Isso significa que o primeiro elemento inserido é o primeiro a sair. Além disso, nós só teremos acesso ao último elemento. Para processar o penúltimo elemento, devemos remover o último



7.3.2 Curiosidade

Uma curiosidade é que tem alguns exemplos interessantes onde a fila é utilizada, são eles:

- Filas de Impressão: Quando você manda vários documentos para imprimir, eles entram numa fila. A impressora processa o primeiro documento que chegou (FIFO).
- Processamento de Tarefas: Em sistemas que recebem muitas solicitações (como um servidor web), as solicitações são colocadas numa fila para serem processadas por ordem de chegada.

- Mensagens (Chat): As mensagens que você recebe num chat geralmente são exibidas na ordem em que chegaram.

7.3.3 Implementação

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

struct elemento{
    int conteudo;
    struct elemento *prox;
};

typedef struct elemento Elem;

struct fila{
    struct elemento *inicio;
    struct elemento *final;
    int qtd;
};

typedef struct fila Fila;

Fila* cria_fila(){
    Fila* fi = (Fila*) malloc(sizeof(Fila));
    if( fi!=NULL){
        fi->final = NULL;
        fi->inicio = NULL;
        fi->qtd = 0;
    }
    return fi;
}

void libera_fila(Fila* fi){
    if( fi!=NULL){
        Elem* no;
        while( fi->inicio!=NULL){
            no = fi->inicio;
            fi->inicio=fi->inicio->prox;
            free(no);
        }
        free(fi);
    }
}

int insere_Fila(Fila* fi ,int x ){
    if( fi==NULL){
```

```

        return 0;
    }
    Elemt* no = (Elemt*) malloc( sizeof(Elemt) );
    if (no==NULL){
        return 0;
    }
    no->conteudo = x;
    no->prox = NULL;

    if ( fi->inicio ==NULL){
        fi->inicio = no;
    } else {
        fi->final->prox = no;
    }
    fi->final= no;
    fi->qtd++;
    return 1;
}

int remove_Fila(Fila* fi){
    if ( fi==NULL){
        return 0;
    }
    if ( fi->inicio==NULL){
        return 0;
    }
    Elemt *no = fi->inicio;
    fi->inicio = fi->inicio->prox;

    if ( fi->inicio==NULL){
        fi->final=NULL;
    }

    free(no);
    fi->qtd--;

    return 1;
}

void imprime_Fila(Fila* fi){
    if ( fi==NULL){
        return;
    }
    Elemt* no =fi->inicio ;
    printf("\n-----Fila-----\n");

```

```

        while( no!=NULL){
            printf("\t %d ", no->conteudo);
            no = no->prox;
        }
        printf("\n-----");
    }
int main(void) {
    Fila* fi = cria_fila();

    insere_Fila(fi , 0);
    insere_Fila(fi , 1);
    insere_Fila(fi , 2);
    insere_Fila(fi , 3);
    insere_Fila(fi , 4);

    imprime_Fila( fi );

    remove_Fila( fi );
    remove_Fila( fi );
    remove_Fila( fi );
    imprime_Fila( fi );

    return 0;
}

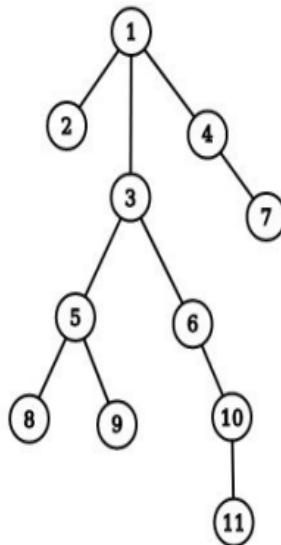
```

Capítulo 8

Arvores

8.1 Conceituando

A arvore na programação é uma estrutura de dados não linear e hierárquica que é usada para representar e organizar dados. Elas são definidas com um conjunto de nós e arestas (que fazem a conexão entre dois nós).



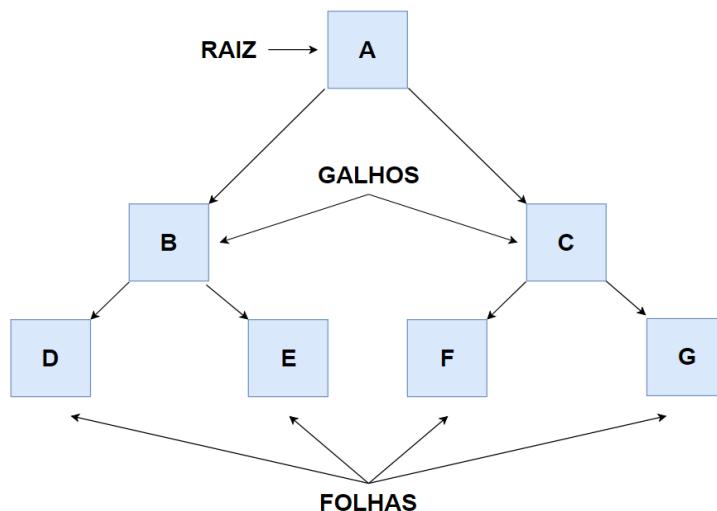
Existem diferentes tipos de arvores na computação, desenvolvidas para diferentes tipos de aplicação, são elas:

- Arvores binária de busca
- Árvore AVL

- Árvore Rubro-Negra
- Árvore B+
- Árvore 2-3
- Árvore 2-3-4
- Quadtree
- Octree

8.1.1 Detalhando a estrutura

Detalhando um pouco a estrutura das árvores, nós temos alguns conceitos essenciais. Veja a imagem abaixo:



- Nó raiz: é o nó mais alto
- Nós internos (galhos): São os nós com filho
- Nós folha (ou externos): São os nós sem filho

Além disso, há um detalhe de hierarquia, chamamos de:

- Nó pai: Antecessor imediato do nó específico que estamos olhando
- Nó filho: Sucessor imediato do nó específico que estamos olhando

8.1.2 Curiosidade

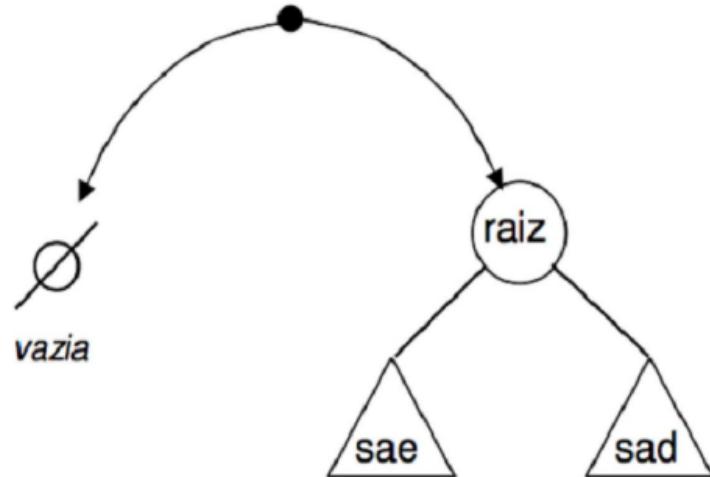
Uma curiosidade é que as árvores estão mais perto de nós do que se imagina. Alguns exemplos da aplicação de árvores no mundo real são:

- Estrutura de diretórios e arquivos de um sistema operacional
- Estrutura usada por diversos algoritmos

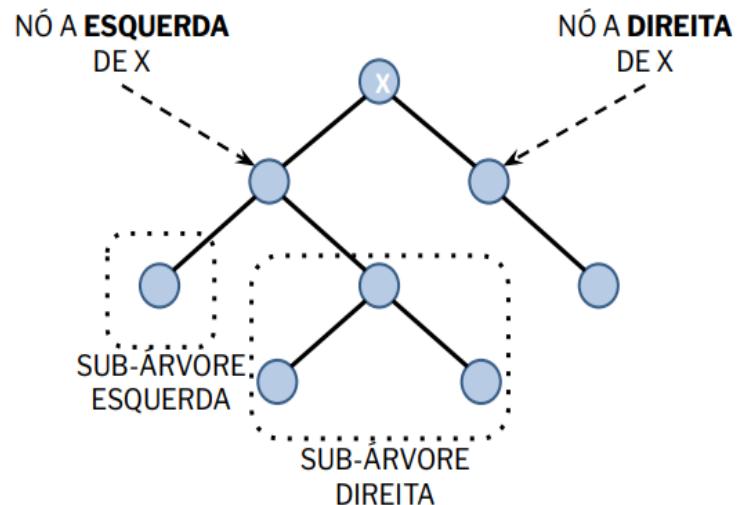
8.2 Arvores binárias

8.2.1 Conceito

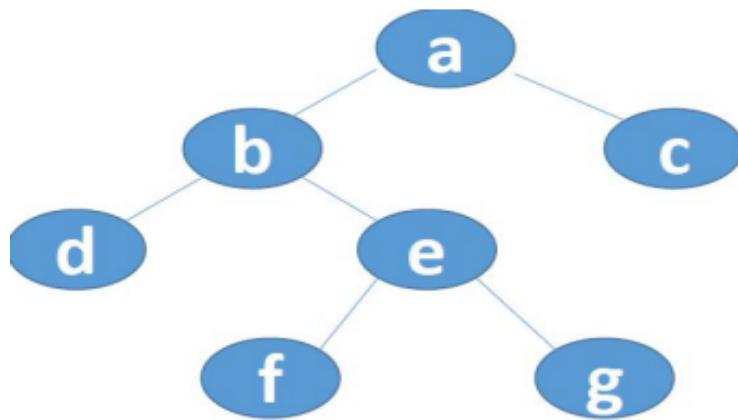
Uma árvore binária é uma árvore em que cada nó tem 0, um ou dois filhos.



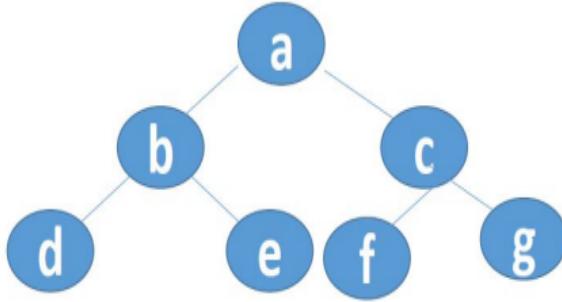
Tendo dois filhos, teremos a subárvore da esquerda e a subárvore da direita.
Veja a imagem a seguir:



Uma árvore pode ser completa se todo elemento que não for folha tem associado exatamente duas subárvore. Ou seja, ou este elemento tem duas subárvore ou não tem nenhuma. Veja uma imagem abaixo:



Agora, uma árvore binária sera cheia se for completa e todas suas folhas estiverem no mesmo nível. Veja a imagem a seguir:



8.2.2 Propriedades

Algumas propriedades das árvores binárias:

- A altura (h) de uma árvore binária é a longitude do maior caminho. Ou o números de vezes que se aplica a relação pai/filho no maior caminho(igual ao número de ramos/galhos).
- O nível de um elemento dentro da árvore binária se define como a longitude do caminho que parte da raiz e chega até esse elemento. Desta forma, o nível da raiz é 0, e o nível de qualquer elemento é o nível de seu pai mais 1.
- Nem sempre existe um caminho entre dois elementos de uma árvore, mas se existe, este caminho é único
- Um caminho entre dois elementos e_1 e e_2 é uma sequencia
- A altura de um nó folha é 0. Pode se considerar que a altura de uma árvore vazia é -1, para facilitar na implementação
- O peso de uma árvore é o número de elementos desta árvore. Recursivamente se pode definir como a soma dos pesos das subárvores mais 1
- De acordo com a definição

Dessa forma, podemos calcular o número de nós de uma árvore cheia a partir de sua altura. Para isso, basta usar a fórmula:

$$N = 2^{(h+1)} - 1$$

Veja o exemplo a seguir:

Por outro lado, se a árvore estiver degenerada (os nós internos tem somente uma subárvore), usaremos a fórmula:

$$N = h + 1$$

Um detalhe que é importante ressaltar é que o esforço computacional necessário para alcançar qualquer nó da arvore será

Uma curiosidade é que tem algumas nomenclaturas que podemos dar para as árvores a depender do seu grau de semelhança com outras árvores:

- Arvores iguais: Duas árvores binarias são iguais se ambas são vazias, ou se suas raízes são iguais, e o mesmo acontece para suas respectivas subárvores esquerda e direita.
- Arvores isomorfas: Duas árvores são isomorfas se têm a mesma forma(estrutura), mas não necessariamente os mesmos elementos
- Arvores semelhantes: Duas árvores são semelhantes se têm os mesmos elementos, ainda que não sejam isomorfas.

Percorso

Sobre o percurso em uma árvore binária, cada nó é visitado uma vez, não existe uma ordem natural para se percorrer os nós de uma árvore binária e percorrer os nós de uma árvore binária pode ser feito para executar uma ação em cada nó.

Podemos percorre-la de 3 formas diferentes:

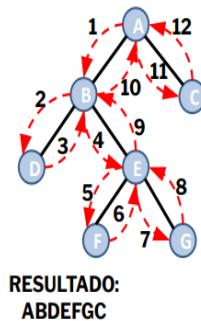
- Percurso pré-ordem: Visita a raiz, o filho da esquerda e o filho da direita
- Percurso um-ordem: Visita o filho da esquerda, a raiz e o filho da direita
- Percurso pós-ordem: Visita o filho da esquerda, o filho da direita e a raiz

Existem outras formas de percursos. Acima, temos os percursos mais importantes. Veja uma imagem abaixo do percurso PRÉ-ORDEM:

Percorso pré-ordem

Ordem de visitação

- Raiz
- Filho esquerdo
- Filho direito



inicia no nó A	
1	imprime A, visita B
2	imprime B, visita D
3	imprime D, volta para B
4	visita E
5	imprime E, visita F
6	imprime F, volta para E
7	visita G
8	imprime G, volta para E
9	volta para B
10	volta para A
11	visita C
12	imprime C, volta para A

Agora veja uma imagem do percurso UM-ORDEM:

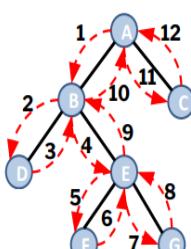
Percorso em-ordem

■ Ordem de visitação

- Filho esquerdo

- Raiz

- Filho direito



RESULTADO:
DBFEGAC

inicia no nó A	
1	visita B
2	visita D
3	imprime D, volta para B
4	imprime B, visita E
5	visita F
6	imprime F, volta para E
7	imprime E, visita G
8	imprime G, volta para E
9	volta para B
10	volta para A
11	imprime A, visita C
12	imprime C, volta para A

Agora veja, por último, a do percurso PÓS-ORDEM:

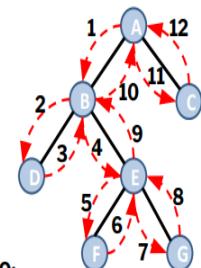
Percorso pós-ordem

■ Ordem de visitação

- Filho esquerdo

- Filho direito

- Raiz



RESULTADO:
DFGEBCA

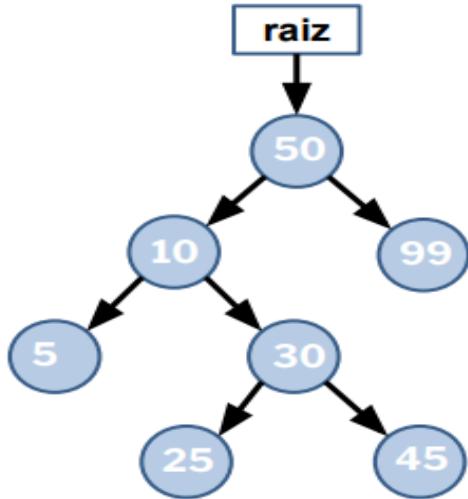
inicia no nó A	
1	visita B
2	visita D
3	imprime D, volta para B
4	visita E
5	visita F
6	imprime F, volta para E
7	visita G
8	imprime G, volta para E
9	imprime E, volta para B
10	imprime B, volta para A
11	visita C
12	imprime C, volta para A e imprime A

8.3 Arvore binaria de busca

Uma arvore binaria de busca é uma árvore binária com uma regra fundamental.

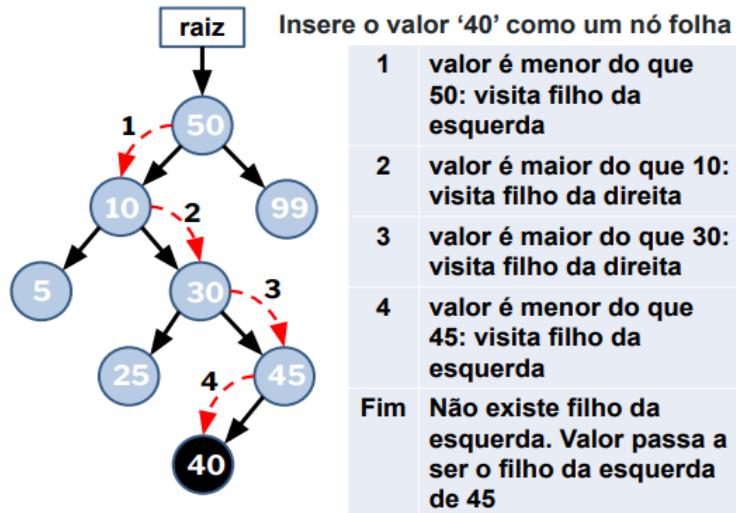
- tudo na subarvore à esquerda é menor que ele
- tudo na subarvore à direita é maior que ele

Veja um exemplo abaixo:



8.3.1 Operações básicas

Inserção

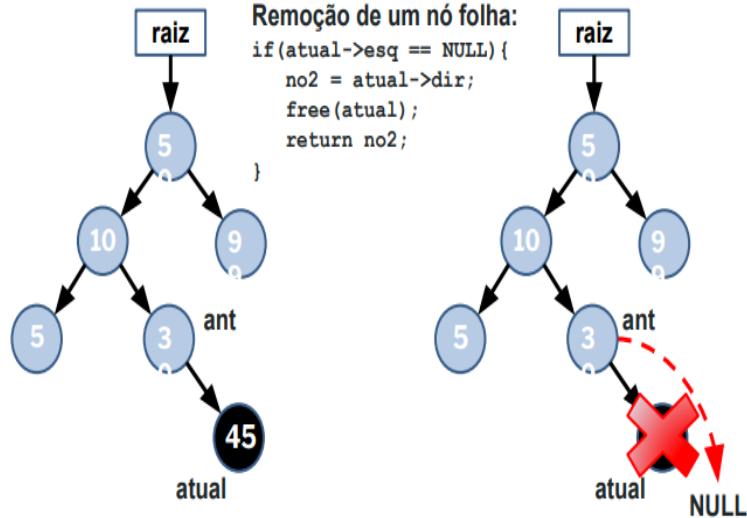


Remoção

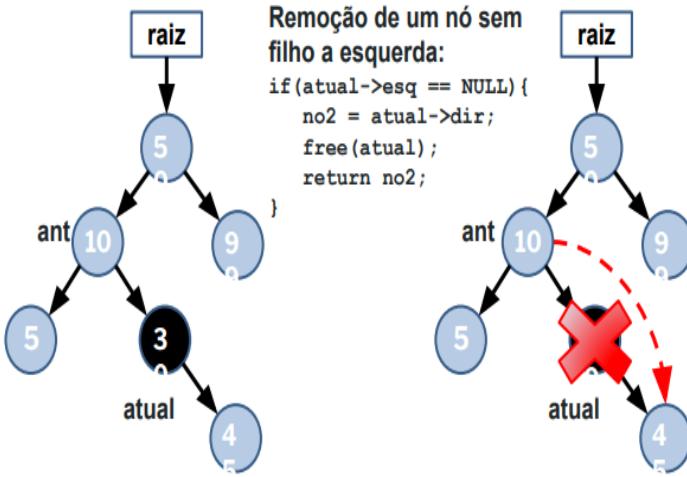
Remover um nó de uma árvore binária de busca não é uma tarefa tão simples quanto a inserção. Isso ocorre porque precisamos procurar o nó a ser removido da árvore o qual pode ser um nó folha ou um nó interno e se for um nó interno,

precisamos reorganizar a árvore para que ela continue sendo uma árvore binária de busca

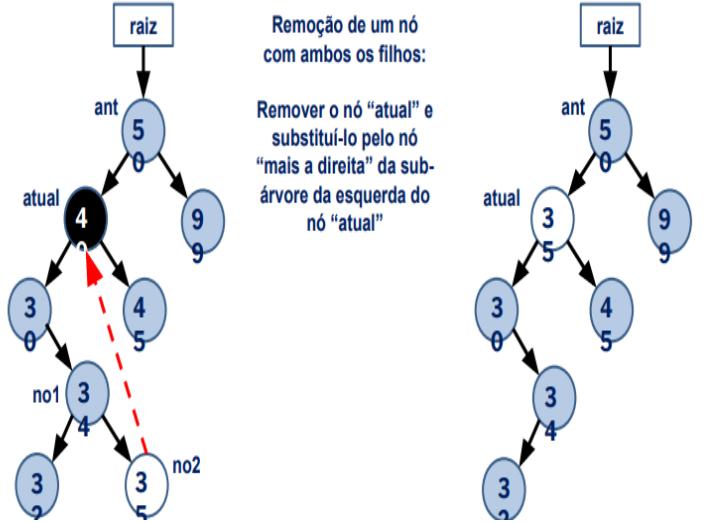
Veja um exemplo de remoção de um NÓ FOLHA:



Agora veja um exemplo de remoção de um nó com 1 filho:



Por fim, veja um exemplo de remoção de um nó com 2 filhos



8.4 Arvores AVL

Infelizmente, os algoritmos de inserção e remoção em árvores binárias não garantem que a árvore gerada a cada passo esteja balanceada. Dependendo da ordem em que os dados são inseridos na árvore, podemos criar uma árvore na forma de uma escada. Veja a imagem abaixo:

A solução para isso foi modificar as operações de inserção e remoção de modo a balancear a árvore a cada inserção e remoção, garantindo assim que a diferença de alturas das sub-árvores esquerda e direita de cada nó seja de no máximo uma unidade

Assim, surgiram as árvores平衡adas, sendo elas:

- AVL
- 2-3-4
- Rubro-Negra

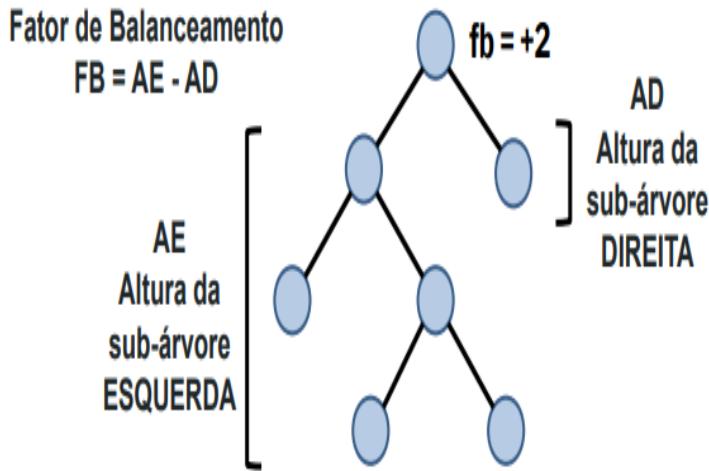
Por aqui, vamos tratar da AVL.

8.4.1 Conceito

A arvore AVL permite o rebalanceamento local da árvore. Apenas a parte afetada pela inserção ou remoção é rebalanceada. Em sua etapa de rebalanceamento, ela usa rotações simples ou duplas, que são executadas a cada inserção ou remoção e buscam manter a árvore binária como uma árvore quase completa

8.4.2 Funcionamento

Assim, nós teremos as rotações que buscarão corrigir o fator de balanceamento (ou fb), que é a diferença entre a altura da subárvore à esquerda - a altura da subárvore à direita. Veja a imagem abaixo:



Acima, perceba que o fator de balanceamento é igual a 2. Isso acontece porque a altura da árvore da esquerda é 3 e a altura da árvore à direita é 1, ou seja $3 - 1 = 2$. Dessa forma, perceba que precisamos平衡ar a árvore, uma vez que o fator de balanceamento deve ser 1, 0 ou -1. Para isso, vamos aplicar as rotações

8.4.3 Rotações

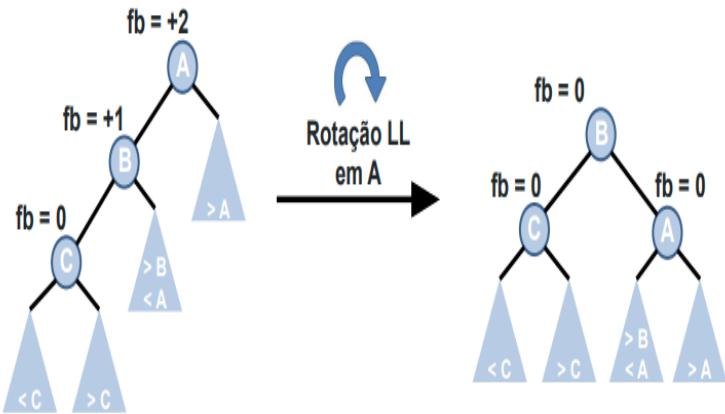
As rotações diferem entre si pelo sentido da inclinação entre o nó pai e filho

- Rotação simples: O nó desbalanceado (pai), seu filho e o seu neto estão todos no mesmo sentido de inclinação. Existe a rotação simples à direita (LL) e rotação dupla à esquerda (RR)
- Rotação dupla: O nó desbalanceado (pai) e seu filho estão inclinados no sentido inverso ao neto (equivale a duas rotações simples). Existe a rotação dupla à direita (LR) e rotação dupla à esquerda (RL)

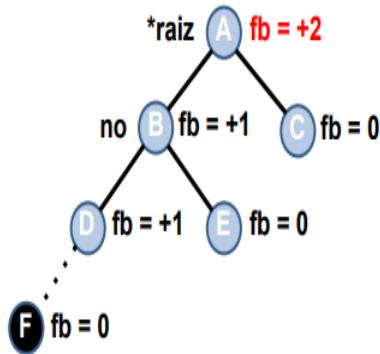
Rotação LL (simples à direita)

Considerando um exemplo onde A é o nó desbalanceado, o que acontece é que um novo nó é inserido na subárvore da esquerda do filho esquerdo de A

Ou seja, é necessário fazer uma rotação à direita, de modo que o nó intermediário B ocupe o lugar de A, e A se torne a subárvore direita de B. Veja abaixo:



Agora veja outro exemplo:



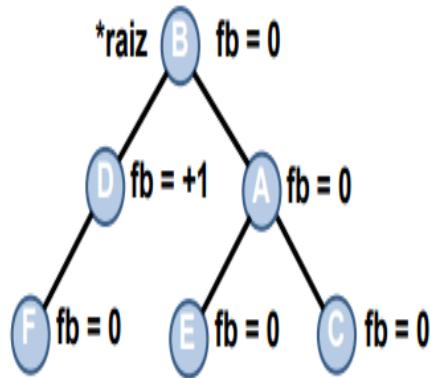
Inserção do nó F na árvore

Árvore fica desbalanceada
no nó A.

Aplicar Rotação LL no nó A

```
no = (*raiz)->esq;
(*raiz)->esq = no->dir;
no->dir = *raiz;
*raiz = no;
```

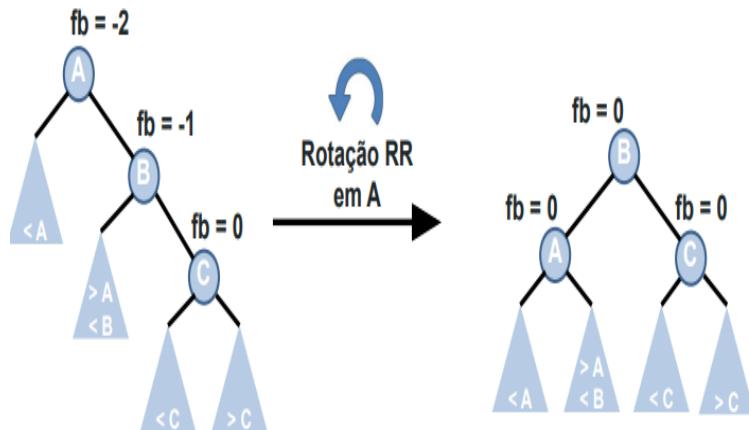
A árvore balanceada:



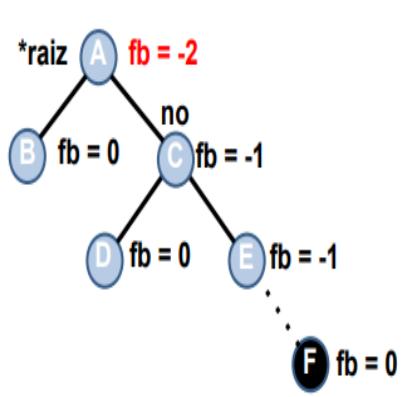
Rotação RR (simples à esquerda)

Considerando um exemplo onde A é o nó desbalanceado, o que acontece é que um novo nó é inserido na sub-árvore da direita do filho direito de A.

Ou seja, é necessário fazer uma rotação à esquerda, de modo que o nó intermediário B ocupe o lugar de A, e A se torne a sub-árvore esquerda de B. Veja abaixo:



Agora veja um outro exemplo:



Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

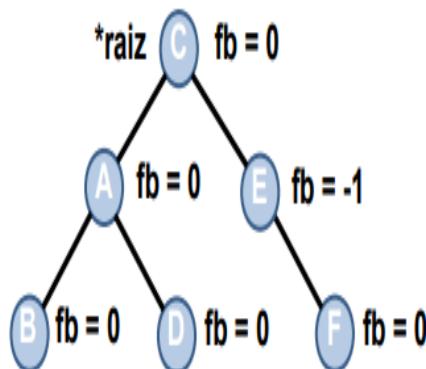
Aplicar Rotação RR no nó A

```

no = (*raiz)->dir;
(*raiz)->dir = no->esq;
no->esq = (*raiz);
(*raiz) = no;

```

A árvore balanceada



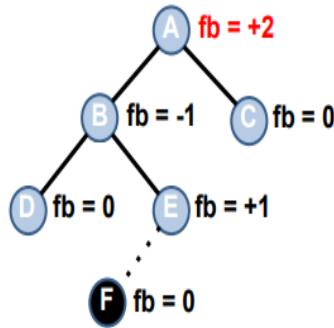
Rotação LR (rotação dupla à direita)

Suponhamos que um novo nó é inserido na sub-árvore da direita do filho esquerdo de A, assim, o nó A fica desbalanceado.

Ou seja, precisamos balancear isso, mas para isso, vamos utilizar um movimento para esquerda e outro para direita (LEFT, RIGHT), utilizando assim a rotação dupla, de modo que o nó C se torne o pai dos nós A (filho da direita) e B (filho da esquerda).

Assim, vamos utilizar um rotação RR (à esquerda) em B e uma rotação LL (à direita) em A. Veja abaixo:

Veja um outro exemplo abaixo:



Inserção do nó F na árvore

Árvore fica desbalanceada no nó A.

Aplicar Rotação LR no nó A.

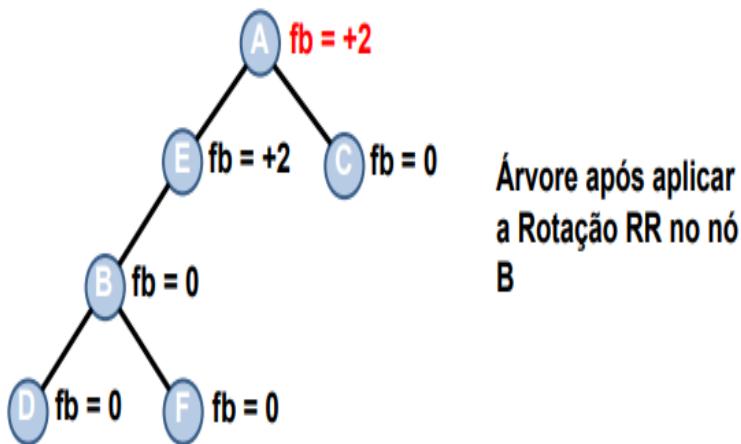
Isso equivale a:

- Aplicar a Rotação RR no nó

B

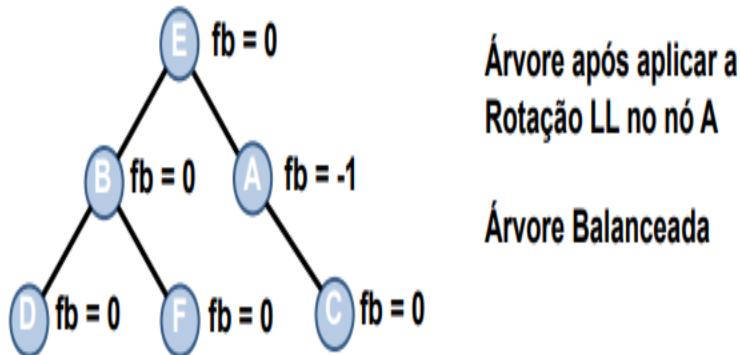
- Aplicar a Rotação LL no nó A

Árvore depois de aplicar a rotação RR no nó B:



Árvore após aplicar
a Rotação RR no
nó B

Árvore depois de aplicar a rotação LL no nó A e por consequencia balanceando a árvore:



Árvore após aplicar a
Rotação LL no nó A

Árvore Balanceada

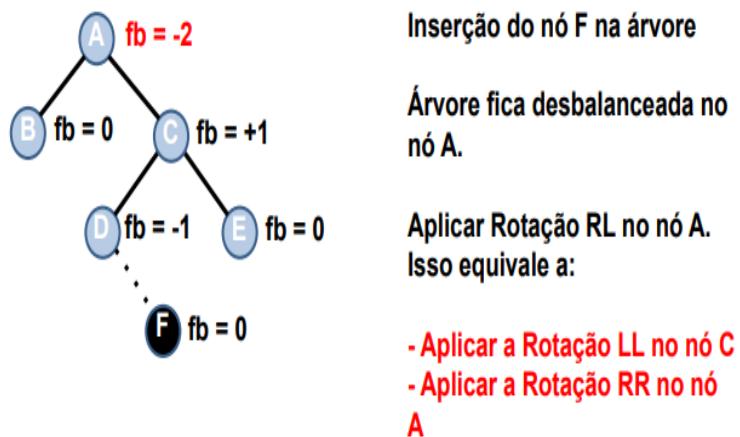
Rotação RL

Suponhamos que um novo nó é inserido na sub-árvore da esquerda do filho direito de A, assim, o nó A fica desbalanceado.

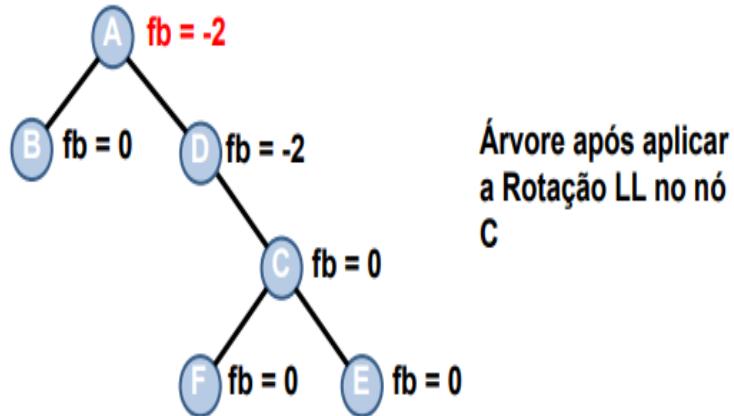
Ou seja, precisamos balancear isso, mas para isso, vamos utilizar um movimento para direita e outro para esquerda (RIGHT, LEFT), utilizando assim a rotação dupla, de modo que o nó C se torne o pai dos nós A (filho da esquerda) e B (filho da direita)

Assim, vamos utilizar um rotação LL (à direita) em B e uma rotação RR (à esquerda) em A. Veja abaixo:

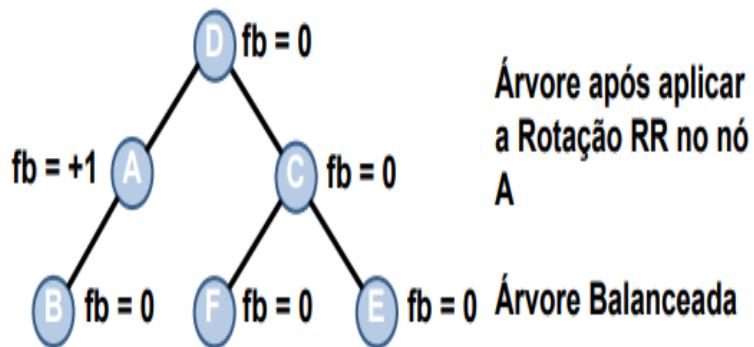
Veja um outro exemplo abaixo:



Árvore depois de aplicar a rotação LL no nó C:



Árvore depois de aplicar a rotação RR no nó A e por consequencia balanceando a árvore:



Macete para as rotações

Fator de Balanceamento de A	Fator de Balanceamento de B	Posições dos nós B e C em relação ao nó A	Rotação
+2	+1	B é filho à esquerda de A C é filho à esquerda de B	LL
-2	-1	B é filho à direita de A C é filho à direita de B	RR
+2	-1	B é filho à esquerda de A C é filho à direita de B	LR
-2	+1	B é filho à de direita A C é filho à esquerda de B	RL

Capítulo 9

Tabela Hash

9.1 Conceito

A Tabela Hash (ou Mapa Hash) é uma estrutura de dados fundamental na programação, projetada para armazenar dados de forma que a busca, inserção e remoção sejam extremamente rápidas (em tempo médio de $O(1)$).

Ela veio para tentar solucionar o problema onde suponhamos que nós estamos guardando dados em um array ou uma lista e queiramos buscar um item, o grande problema/limitação é:

- Precisamos olhar item por item. Isso pode ser muito lento e esse é um processo $O(n)$

Outro detalhe é que as árvores binárias de busca até resolvem esse problema, mas de forma parcial, uma vez que:

- Podem ficar desequilibradas
- Podem perder eficiência
- Mesmo no melhor caso, fazem buscas em $O(\log n)$

Dessa forma, faltava uma estrutura que permitia

- buscar muito rápido, sem percorrer tudo
- em tempo constante, $O(1)$
- sem depender de ordenação
- sem depender de estrutura balanceada

E foi assim que surgiu a tabela hash, uma estrutura extremamente eficiente para buscas, inserções e remoções

9.2 Funcionamento

A ideia central é transformar uma chave (ex: nome, número, CPF, texto) em um índice de array. Suponha que você tenha um array interno com vários espaços. Exemplo:

```
[ __, __, __, __, __, __, __, __ ]
```

Para você saber onde colocar ou encontrar algo, você usa uma função de hash. Essa função:

- recebe a chave
- aplica algum cálculo
- devolve um número
- esse número é usado como posição no array

Exemplos simples:

Exemplo simples:

```
bash Copiar código  
  
hash("João") → retorna 5  
hash("Maria") → retorna 2  
hash("Pedro") → retorna 7
```

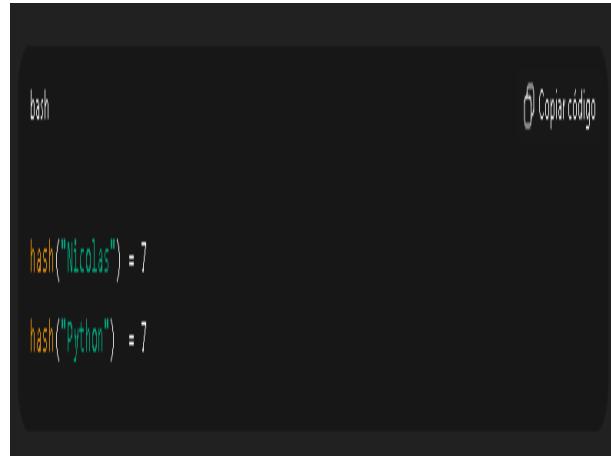
Então dentro da tabela hash fica assim:

```
arduino Copiar código  
  
índice 2 → "Maria"  
índice 5 → "João"  
índice 7 → "Pedro"
```

Isso é extremamente eficiente, pois ao buscar por "João", você roda a hash da palavra "João" e ela te devolve 5, ai você vai diretamente no indice 5 e encontra o que você quer

9.3 As colisões

Um problema que pode acontecer são duas chaves gerar o mesmo hash. Exemplo:



A screenshot of a terminal window with a dark background. In the top right corner, there is a button labeled "Copiar código" with a copy icon. The terminal displays two lines of Python code: "hash('Nicolas') = 7" and "hash('Python') = 7".

```
bach
hash('Nicolas') = 7
hash('Python') = 7
```

Isso pode acontecer porque a tabela hash tem tamanho limitado, mas você tem infinitas chaves possíveis