

# MC202: Estruturas de Dados

Este documento reúne uma compilação de anotações pessoais, esquemas e resumos desenvolvidos ao longo do semestre da disciplina de Estrutura de Dados (MC202), oferecida pelo Instituto de Computação da Unicamp.

O conteúdo deste pdf tem como base fundamental o material didático, as explicações em sala e, principalmente, os slides apresentados pelo **Prof. Rafael C. S. Schouery**. A estrutura dos tópicos segue a didática proposta pelo professor. Vale ressaltar que, embora baseado no material oficial, quaisquer erros de interpretação ou implementação dos códigos são de minha responsabilidade.

A disciplina de Estruturas de Dados é um dos pilares centrais da Ciência e Engenharia de Computação. Mais do que ensinar a linguagem C, o curso foca na eficiência e na organização da memória. Ao longo destas notas, aprendi como manipular dados de forma inteligente — saindo da alocação básica de vetores e listas ligadas, passando pelas árvores binárias e heaps, até chegar à modelagem de problemas reais através de grafos e tabelas hash.

## 1. Curso de C (parte 1):

Para começar a comparar as duas linguagens, utilizaremos como base o código ao lado escrito em Python. Para C temos as seguintes diferenças:

### a) funções:

Em Python	Em C
<pre>1 def maximo(a, b): 2     if a &gt; b: 3         return a 4     else: 5         return b</pre>	<pre>1 int maximo(int a, int b) { 2     if (a &gt; b) { 3         return a; 4     } else { 5         return b; 6     } 7 }</pre>

Em C, uma função é declarada como: tipo **nome (tipo parametro1, tipo parametro2, ...);**

A linguagem C é **estaticamente** tipada, os tipos das variáveis estão definidos no código, ao contrário de Python que é **dinamicamente** tipada, em C temos int, float, double, char, etc.

O tipo **int** armazena números inteiros, usualmente de 32 bits, mas depende do compilador, algumas operações com inteiros:

### b) Blocos:

Em python começam com **:** e são indentados, já em C os blocos são delimitados por **{ }**, indentação não é obrigatória, mas é uma boa prática de programação.

A maioria das linhas em C são terminadas em **;**, blocos são exceção.

Algumas operações	
a + b	soma
a - b	subtração
a * b	multiplicação
a / b	divisão inteira, i.e., 8 / 5 é 1
a % b	resto da divisão, i.e., 8 % 5 é 3
a += b	o mesmo que a = a + b
a -= b	o mesmo que a = a - b
a *= b	o mesmo que a = a * b
a /= b	o mesmo que a = a / b
a %= b	o mesmo que a = a % b
a++	o mesmo que a += 1
++a	o mesmo que a += 1
a--	o mesmo que a -= 1
--a	o mesmo que a -= 1

Algumas operações em C também são diferentes segue a tabela ao lado.

```
1 def maximo(a, b):
2     if a > b:
3         return a
4     else:
5         return b
6
7 def potencia(a, b):
8     prod = 1
9     for i in range(b):
10        prod = a * prod
11    return prod
12
13 print("Entre com a e b")
14 a = int(input())
15 b = int(input())
16 maior = maximo(a, b)
17 pot = potencia(a, b)
18 print("Maior:", maior)
19 print("a^b:", pot)
```

c) Condicionais:

Em C temos 3 opções de **if**:

```
1 if (condicao) {  
2     ...  
3 }
```

```
1 if (condicao) {  
2     ...  
3 } else {  
4     ...  
5 }
```

```
1 if (condicao) {  
2     ...  
3 } else if (condicao) {  
4     ...  
5 } else {  
6     ...  
7 }
```

d) Variáveis:

Em Python

```
1 def potencia(a, b):  
2     prod = 1  
3     for i in range(b):  
4         prod = a * prod  
5     return prod
```

Em C

```
1 int potencia(int a, int b) {  
2     int i, prod = 1;  
3     for (i = 0; i < b; i++) {  
4         prod = a * prod;  
5     }  
6     return prod;  
7 }
```

Em Python, uma variável é declarada automaticamente, basta atribuir para ela ou definir-la como parâmetro. Em C, a variável precisa ser declarada antes do uso, fazemos isso no início da função.

- **int i;** declara uma variável de nome **i** do tipo **int**.
- **int i, prod = 1;** declara **i** e **prod** do tipo **int**, com **prod** tendo valor inicial 1;

e) Laços:

Em Python

```
1 def potencia(a, b):  
2     prod = 1  
3     for i in range(b):  
4         prod = a * prod  
5     return prod
```

Em C

```
1 int potencia(int a, int b) {  
2     int i, prod = 1;  
3     for (i = 0; i < b; i++) {  
4         prod = a * prod;  
5     }  
6     return prod;  
7 }
```

Em C não há **for ... in**, mas temos **while**, **do ... while** e **for**.

```
1 while (condicao) {  
2   ...  
3 }
```

```
1 do {  
2   ...  
3 } while (condicao);
```

```
1 for (inicializacao; condicao; atualizacao) {  
2   ...  
3 }
```

O **while** executa enquanto **condicao** for verdadeiro.

**Do while** executa o bloco enquanto a **condicao** for verdadeira, mas sempre executa a primeira vez.

**For** é um comando com 3 partes, a **inicialização** é executada apenas na primeira vez, a **condição** é usada na primeira vez, depois da inicialização e se for falsa o laço para. A **atualização** é usada antes de testar condição.

f) Função main:

Em Python

```
1 print("Entre com a e b")  
2 a = int(input())  
3 b = int(input())  
4 maior = maximo(a, b)  
5 pot = potencia(a, b)  
6 print("Maior:", maior)  
7 print("a^b:", pot)
```

Em C

```
1 int main() {  
2   int a, b, maior, pot;  
3   printf("Entre com a e b\n");  
4   scanf("%d %d", &a, &b);  
5   maior = maximo(a, b);  
6   pot = potencia(a, b);  
7   printf("Maior: %d\n", maior);  
8   printf("a^b: %d\n", pot);  
9   return 0;  
10 }  
11  
12 int maximo(int a, int b);
```

Em C, a execução  
do programa

começa pela função main que sempre devolve um int:

Se devolver 0 significa que não houve erros, valores diferentes indicam que um erro aconteceu:

g) Impressão:

A impressão no C é feita pela função **printf**: O **%d** significa substituir por um inteiro (existem outras substituições **%f**, **%s**, etc.), ele recebe um parâmetro com a string a ser impressa e um parâmetro adicional para cada %. A substituição é feita da esquerda para a direita da string e ele não adiciona automaticamente a quebra de linha, como o Python fazia, precisa digitar “**\n**”.

#### h) Leitura:

Para leitura faremos algo muito parecido. A leitura no C é feita pela função **scanf**: String diz quantos valores serão lidos e os seus tipos, precisa passar o endereço da variável usando operador **&** (Veremos mais sobre isso em breve, por enquanto não esqueça do **&**). Ele ignora espaços em branco, tabs e quebras de linhas.

Se queremos dois números como input do usuário: `scanf("%d %d", &num1, &num2);`

#### i) O programa inteiro:

No começo, colocamos as bibliotecas a serem usadas. Usamos a biblioteca stdio.h por causa de printf e scanf. Para utilizar alguma biblioteca em C devemos usar:

`#include <nome>`

#### j) Executando o programa:

Python é uma linguagem interpretada, C é compilada. O interpretador do Python abre e executa o seu código. Já o do C gera um arquivo executável, depois não depende mais do compilador. Compilando (no terminal):

`gcc -std=c99 -Wall -Werror -g -lm programa.c -o programa`

Flags: Tudo que está com menos na frente.

- ansi: usa o padrão C89(versão mais portável);
- Wall: dá mais warnings de compilação;
- pedantic-errors: força a seguir o padrão ANSI;
- Werror: warnings viram erros de compilação;
- g: permite usar gdb e valgrind;
- lm: permite usar funções matemáticas;
- o: define o nome do programa;

Para executar o programa no terminal é só fazer “`./programa`”.

```
1 #include <stdio.h>
2
3 int maximo(int a, int b) {
4     if (a > b) {
5         return a;
6     } else {
7         return b;
8     }
9 }
10
11 int potencia(int a, int b) {
12     int i, prod = 1;
13     for (i = 0; i < b; i++) {
14         prod = a * prod;
15     }
16     return prod;
17 }
18
19 int main() {
20     int a, b, maior, pot;
21     printf("Entre com a e b\n");
22     scanf("%d %d", &a, &b);
23     maior = maximo(a, b);
24     pot = potencia(a, b);
25     printf("Maior: %d\n", maior);
26     printf("a^b: %d\n", pot);
27     return 0;
28 }
29
30 int maximo(int a, int b);
```

### k) O programa refatorado:

Refatorar significa deixá-lo mais rápido/curto. Aqui foram feitas as seguintes mudanças: Quando o bloco de um if, else, for ou while tiver apenas uma linha, podemos omitir o { e }. Também na linha 13 podemos escrever prod \*= a. Além disso, o printf pode imprimir os resultados de funções

```
1 #include <stdio.h>
2
3 int maximo(int a, int b) {
4     if (a > b)
5         return a;
6     else
7         return b;
8 }
9
10 int potencia(int a, int b) {
11     int prod = 1, i;
12     for (i = 0; i < b; i++)
13         prod *= a;
14     return prod;
15 }
16
17 int main() {
18     int a, b;
19     printf("Entre com a e b\n");
20     scanf("%d %d", &a, &b);
21     printf("Maximo: %d\n", maximo(a, b));
22     printf("Potencia: %d\n", potencia(a, b));
23 }
```

### l) Um programa com listas/vetores:

Em Python

```
1 print("Digite 10 números")
2 lista = []
3 for i in range(10):
4     lista.append(int(input()))
5 print("Positivos")
6 for x in lista:
7     if x > 0:
8         print(x)
```

Em C

```
1 #include <stdio.h>
2
3 int main() {
4     int i, lista[10];
5     printf("Digite 10 números\n");
6     for (i = 0; i < 10; i++)
7         scanf("%d", &lista[i]);
8     printf("Positivos\n");
9     for (i = 0; i < 10; i++) {
10         if (lista[i] > 0)
11             printf("%d\n", lista[i]);
12     }
13     return 0;
14 }
```

Em C, as listas são bem diferentes em relação ao Python:

- São chamadas de **vetores(arrays)**;
- Todos os elementos devem ser de um mesmo tipo;
- Têm tamanho fixo definido na declaração da variável;
- Exemplo de declaração: **int lista [10]** (cria uma lista de 10 **ints**).

Cada **lista[i]** é um **int**:

- Para imprimir **lista[i]**:
- Printf("%d", lista[i]);**

- Para ler um número e guardar em **lista[i]**:
- scanf("%d", &lista[i]);**

Para refatorar esse código, podemos ter uma função que lê vetores e ter outra função que imprime apenas os positivos:

A função é do tipo **void**. Isto é a função não devolve um valor, ela recebe um vetor chamado **lista**: Não precisamos especificar o tamanho entre o [] (ele é apenas um parâmetro) e é nossa responsabilidade saber o tamanho do vetor (por isso precisamos do parâmetro n, não há equivalente ao len() do Python).

```
1 void imprime_positivos(int lista[], int n) {  
2     int i;  
3     printf("Positivos\n");  
4     for (i = 0; i < n; i++)  
5         if (lista[i] > 0)  
6             printf("%d\n", lista[i]);  
7 }
```

Em C não é possível devolver um vetor... Portanto, para lê-lo precisamos passá-lo como parâmetro e modificar seu conteúdo.

```
1 void le_vetor(int lista[], int n) {  
2     int i;  
3     printf("Digite %d números\n", n);  
4     for (i = 0; i < n; i++)  
5         scanf("%d", &lista[i]);  
6 }
```

### Cuidados com vetores em C:

- A responsabilidade de acessar apenas posições válidas é sua: Se você declarou um vetor com 10 posições e tentar acessar a posição 10, 11, 12... ou você terá um erro de execução (**segmentation fault**) ou não... (pode imprimir o valor de outra variável ou mudar o valor dela).
- No C, um vetor é um bloco contíguo de memória e o C assume que você usará o bloco corretamente, logo não há checagem dos limites do vetor. O que ocorre muitas vezes é **off-by-one** (se o vetor tem n posições, você não deve acessar a posição n).

## **2. Curso de C (parte 2):**

### a) Exemplo de programa em C para o cálculo de raízes quadradas:

Para isso usaremos o método babilônico, seja  $y_1$  uma estimativa para  $y = \sqrt{x}$ , por exemplo  $y = x$ , é notório que quanto melhor a estimativa mais rápida o programa. Faça  $y_n = (\frac{y_{n-1} + x}{y_{n-1}})$ , se  $|y_n - y_{n-1}|$  for “grande”, volte para o  $y_n$  e no final o programa deve devolver  $y_n$ .

Em Python sua representação seria dessa forma:

```

1 ERRO = 1e-12
2
3
4 def square_root(x):
5     y = x
6     erro_pequeno = False
7     while not erro_pequeno:
8         anterior = y
9         y = (y + x / y) / 2
10        if abs(anterior - y) <= ERRO:
11            erro_pequeno = True
12    return y
13
14
15 print("Entre com o numero:")
16 x = float(input())
17 print("Raiz quadrada:", square_root(x))

```

O código em C:

- A biblioteca math.h contém várias funções matemáticas;
- Em C as funções recebem/devolvem parâmetros/resultados de um tipo específico;
- Tipo podem ser convertidos: valor int pode ser convertido para double, por exemplo escreva (double) x e 1 será convertido para 1.0;
- Temos duas funções diferentes que calculam valor absoluto: int abs(int x) e double fabs(double x): abs(-7.9) é 7 e fabs(-3) é 3.0. Os valores são convertidos automaticamente.

-A diretiva #define cria uma macro, onde aparecer a palavra ERRO, substitua por 1e-12;

	Python	C
6 / 4	1.5	1
6.0 / 4.0	1.5	1.5
6.0 / 4	1.5	1.5
6 / 4.0	1.5	1.5
6 // 4	1	
6.0 // 4.0	1.0	Op. não existe
6.0 // 4	1.0	
6 // 4.0	1.0	
6 % 4	2	2
6.0 % 4.0	2.0	erro de compilação
6.0 % 4	2.0	erro de compilação
6 % 4.0	2.0	erro de compilação

Para traduzir para C, precisamos primeiro entender a diferença entre os tipos float e double. O tipo float é um número de ponto flutuante com precisão simples, em geral usa 32 bits, sua leitura e escrita é feita com %f, se usar notação científica %e, ou se quiser o mais curto %g. O tipo double é um número de ponto flutuante com precisão dupla, em geral usa 64 bits, gastando mais memória e sendo mais lento (mais usado em geral), sua leitura/impressão é feita com %lf, %le ou %lg.

```

1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }

```

- Divisão real e inteira:

Se necessário fizemos casting, se x vale 6 e y vale 4, então (double)x/y é 1.5.

b) Exemplo de códigos:

```
1 #include <stdio.h>
2
3 int main() {
4     double A, B, C, triangulo, circ, trap, quad, reta;
5     scanf("%lf %lf %lf", &A, &B, &C);
6     triangulo = (A * C) / 2;
7     circ = (3.14159 * C*C );
8     trap = ((A+B) * C) / 2;
9     quad = B * B;
10    reta = A * B;
11    printf("TRIANGULO: %.3lf\n", triangulo);
12    printf("CIRCULO: %.3lf\n", circ);
13    printf("TRAPEZIO: %.3lf\n", trap);
14    printf("QUADRADO: %.3lf\n", quad);
15    printf("RETANGULO: %.3lf\n", reta);
16    return 0;
17 }
```

```
1 #include <stdio.h>
2
3 int maior_valor(int a, int b){
4     if ( a < b){
5         return b;
6     } else {
7         return a;
8     }
9 }
10
11 int main() {
12     int A, B, C, maior, maior2;
13     scanf("%d %d %d", &A, &B, &C);
14     maior = maior_valor(A, B);
15     maior2= maior_valor(maior, C);
16     printf("%d eh o maior\n", maior2);
17
18     return 0;
19 }
```

O código recebe três valores e calcula algumas áreas. Note o **scanf** como é feito e no **printf** a estrutura para printar com uma quantidade definida de algarismos, nesse caso com 3.

O código tem uma estrutura simples que recebe três valores inteiros e devolve seu maior. Atentar no uso da função com **if** e **else** e, também, no print com um inteiro no começo da frase.

3. Curso de C (Parte 3):

a) Tipo char:

Uma letra ou caractere em C é representado pelo tipo **char**, é um número inteiro (normalmente com 8 bits), podemos somar, subtrair, multiplicar, dividir como se fosse um int, mas com menos valores válidos. Representamos seus caracteres usando a tabela ASCII. Representamos constantes usando aspas simples:

'a' significa o número do caractere à tabela ASCII, não é como no Python que precisávamos do seu valor.

Para ler e imprimir usamos **%c**.

32	(espaço)	51	3	70	F	89	Y	108	I
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	\$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	_	113	q
38	&	57	9	76	L	95	-	114	r
39	,	58	:	77	M	96	~	115	s
40	(	59	:	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	-
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

b) Comparações e operadores lógicos em C:

Como no Python, os operadores de comparação a seguir: `<=`, `>`, `>=`, `==` e `!=`;

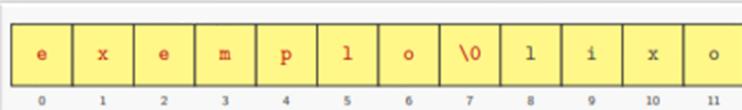
Em C não temos variáveis booleanas como no Python (True e False), o C considera o valor 0 como falso e quaisquer valores diferentes de 0 como verdadeiros.

	Python	C
E	<code>and</code>	<code>&amp;&amp;</code>
Ou	<code>or</code>	<code>  </code>
Não	<code>not</code>	<code>!</code>

Os operadores lógicos são diferentes em C, apresentados na figura ao lado.

c) Strings em C:

Strings em C são vetores de `char` terminados com '`\0`'. Por exemplo, podemos ter um vetor de char com 12 posições, mas a string ter apenas 7 caracteres.



O tamanho da string é o número de caracteres antes do '0';

Função que conta o tamanho de caracteres de uma string:

```
1 int tamanho(char string[]) {
2     int i;
3     for (i = 0; string[i] != '\0'; i++) ;
4     return i;
5 }
```

Lemos strings usando `%s` (Leia até o primeiro espaço em branco) e não colocamos o “`&`” antes do nome da variável.

Lemos strings com espaços usando a função `fgets`: `fgets(parâmetro 1, parâmetro 2, parâmetro 3)`.

-Primeiro parâmetro: nome da variável;

-Segundo parâmetro: tamanho máximo da string (contando o '`\0`');

-Terceiro parâmetro: de qual arquivo devemos ler (se for da entrada padrão `stdin`);

d) Biblioteca string.h:

A biblioteca `string.h` tem várias funções úteis:

- `strlen` devolve o tamanho da string;

- `strcmp` compara duas strings já que não podemos usar (`<=`, `>`, `>=`, `==` e `!=`);

- **strcpy** copia uma string;
- **strcat** concatena duas strings;

### Tipos mais comuns em C:

dado	tipo	formato	ex. de constante
inteiros	int	%d	10
ponto flutuante	float	%f %g %e	10.0f 2e-3f
ponto flutuante (precisão dupla)	double	%lf %lg %le	10.0 2e-3
caractere	char	%c	'c'
string	char []	%s	"string"

Obs: Temos variações de tamanho para int:

- **short** ou **short int** — %hi ( pelo menos 16 bits).
- **long** ou **long int** — %li (pelo menos 32 bits).
- **long long** ou **long long int** — %lli – (pelo menos 64 bits).

#### 4. Curso de C (Parte 4):

##### a) Registros:

Registros são coleções de dados relacionados de vários tipos, organizados em uma única estrutura e referenciados por um nome comum. Cada dado é chamado de membro do registro, sendo cada membro acessado por um nome na estrutura, cada estrutura define um novo tipo com as mesmas características de um tipo padrão da linguagem.

Para declarar uma estrutura com N membros:

```

1 struct identificador {
2   tipo1 membro1;
3   tipo2 membro2;
4   ...
5   tipoN membroN;
6 };

```

Para declarar um registro:

```
struct identificador nome_registro;
```

Ficha de dados cadastrais de um aluno

```

1 struct data {
2   int dia;
3   int mes;
4   int ano;
5 };
6
7 struct ficha_aluno {
8   int ra;
9   int telefone;
10  char nome[30];
11  char endereco[100];
12  struct data nascimento;
13 };

```

Exemplo de ficha de dados cadastrais de aluno.

Para acessar um membro do registro usamos a estrutura registro.membro, no exemplo anterior para imprimir o nome do aluno, usariamos:

```
printf("Aluno: %s\n", aluno.nome);
```

```
1 #include<stdio.h>
2
3 struct data
4 {
5     int dia;
6     int mes;
7     int ano;
8 };
9
10
11 int main()
12 {
13     struct data data_nascimento;
14     scanf("%d %d %d", &data_nascimento.dia, &data_nascimento.mes, &data_nascimento.ano);
15
16     printf("A data digitada foi: %d/%d/%d\n", data_nascimento.dia, data_nascimento.mes, data_nascimento.ano);
17 }
18
19 return 0;
```

Aqui está um programa que lê a data de nascimento como uma estrutura e a printa do mesmo modo, note que declaramos a estrutura **struct data**.

### b) Typedef:

O **typedef** permite dar um novo nome para um tipo. Exemplo:

```
typedef unsigned int u32
```

Com isso é possível declarar uma variável: **u32 x**. Escrever **unsigned int** ou **u32** é a mesma coisa;

Usando o **typedef** para dar nome para a **struct**:

```
1 typedef struct identificador {
2     tipo1 membro1;
3     tipo2 membro2;
4     ...
5     tipoN membroN;
6 } novonome;
```

Com isso ao invés de declarar uma variável dessa forma:

```
struct identificador var;
```

Podemos simplesmente escrever;  
**novonome var**

Para exemplificar vamos escrever um programa que lida com números complexos, somando dois números complexos lidos e calculando seu módulo. →

```
1 #include <stdio.h>
2 #include <math.h>
3
4 typedef struct {
5     double real;
6     double imag;
7 } complexo;
8
9 int main() {
10     complexo a, b, c;
11
12     printf("Digite a parte real e imaginaria do primeiro numero: ");
13     scanf("%lf %lf", &a.real, &a.imag);
14
15     printf("Digite a parte real e imaginaria do segundo numero: ");
16     scanf("%lf %lf", &b.real, &b.imag);
17
18     c.real = a.real + b.real;
19     c.imag = a.imag + b.imag;
20
21     double modulo = sqrt(c.real * c.real + c.imag * c.imag);
22     printf("O modulo da soma eh: %lf\n", modulo);
23
24 }
25 }
```

c) Tipos abstratos de dados (TAD):

Caso quiséssemos utilizar números complexos em vários programas, bastaria copiar a struct e as funções, contudo essa solução não é DRY (don't repeat yourself), para isso quebramos o programa em três partes:

1. Implementação das funções para os números complexos:
  - Definem como calcular soma, absoluto etc.;
  - Chamamos de **Implementação**;
2. Código que utiliza as funções de números complexos:
  - Soma dois números complexos sem se importar como;
  - Calcula o absoluto sem se importar como;
  - Mas precisa conhecer o protótipo das funções...;
  - Chamamos de **Cliente**;
3. Struct e protótipos das funções para números complexos:
  - Define o que o Cliente pode fazer;
  - Define o que precisa ser implementado;
  - Chamamos de **Interface**;

Um TAD é um conjunto de valores associado a um conjunto de **operações permitidas** nesses dados, em C um TAD é declarado como uma struct.

- **Interface**: conjunto de operações de um TAD;
- **Implementação**: conjunto de algoritmos que realizam as operações;
- **Cliente**: código que utiliza/chama uma operação;

Para seguir o exemplo dos números complexos:

1) Primeiro criamos um arquivo complexos.h com a struct e os protótipos de função:

```
1 typedef struct {
2     double real;
3     double imag;
4 } complexo;
5
6 complexo complexo_novo(double real, double imag);
7
8 complexo complexo_soma(complexo a, complexo b);
9
10 double complexo_absoluto(complexo a);
11
12 complexo complexo_le();
13
14 void complexo_imprime(complexo a);
15
16 int complexos_iguais(complexo a, complexo b);
17
18 complexo complexo_multiplicacao(complexo a, complexo b);
19
20 complexo complexo_conjugado(complexo a);
```

É interessante colocar comentários nas funções para dizer o que cada uma delas faz.

2) Segundo criamos um arquivo complexos.c com as implementações:

```
1 #include <stdio.h> ← bibliotecas usadas
2 #include <math.h>
3 #include "complexos.h" ← tem a definição da struct
4
5 complexo complexo_novo(double real, double imag) {
6     complexo c;
7     c.real = real;
8     c.imag = imag;
9     return c;
10}
11
12 complexo complexo_soma(complexo a, complexo b) {
13     return complexo_novo(a.real + b.real, a.imag + b.imag);
14}
15
16 complexo complexo_le() {
17     complexo a;
18     scanf("%lf %lf", &a.real, &a.imag);
19     return a;
20}
```

- <> são para bibliotecas que já vem no C e para procurar na pasta as que eu criei devo escrever entre "".

3) Usando as funções:

```
1 #include <stdio.h>
2 #include "complexos.h" ← tem a struct e as funções
3
4 int main() {
5     complexo a, b, c;
6     a = complexo_le();
7     b = complexo_le();
8     c = complexo_soma(a, b);
9     complexo_imprime(c);
10    printf("%lf\n", complexo_absoluto(c));
11    return 0;
12}
```

Depois no código podemos usar as funções normalmente.

#### d) Compilando:

Seguindo o exemplo anterior podemos ter dúvidas em como compilar já que temos três arquivos diferentes:

- **cliente.c** contém a função **main**;
- **complexos.c** contém a implementação;
- **complexos.h** contém a interface;

Compilando por partes:

- **gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c**  
– Vai gerar o arquivo compilado **cliente.o**
- **gcc -std=c99 -Wall -Werror -Wvla -g -c complexos.c**

- Vai gerar o arquivo compilado **complexos.o**;
  - gcc cliente.o complexos.o -lm -o cliente
  - Faz a linkagem, gerando o executável cliente;
  - Adicionamos **cliente.o** e **complexos.o**;
  - E outras bibliotecas, por exemplo, **-lm**;

Compilando com MAKEFILE:

É mais fácil usar um Makefile para compilar:

```
1 all: cliente
2
3 cliente: cliente.o complexos.o
4 gcc cliente.o complexos.o -g -lm -o cliente
5
6 cliente.o: cliente.c complexos.h
7 gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c
8
9 complexos.o: complexos.c complexos.h
10 gcc -std=c99 -Wall -Werror -Wvla -c complexos.c
```

Assim, basta executar make na pasta com os arquivos.

Obs: Pode ser que você tenha dois tipos abstratos de dados e um precise incluir o outro, o que leva a um loop de inclusão. Podemos usar o **#ifndef** para evitar isso:

```
1 #ifndef ARQUIVO_H // trocamos ARQUIVO pelo nome do arquivo
2 #define ARQUIVO_H
3
4 // Conteúdo do arquivo.h
5
6 #endif
```

## 5. Curso de C (Parte 5):

### a) Ponteiros:

Toda informação utilizada pelo programa está em algum lugar, toda variável tem um endereço de memória, cada posição de um vetor e membro de um registro também.

Um ponteiro é uma variável que armazena um **endereço** para um tipo específico de informação (int, char, double, structs declaradas).

Exemplo:

- `int *p;` declara um ponteiro para `int`;

- Seu nome é `p`;
- Seu tipo é `int *`;
- Armazena um endereço de um `int`;

Assim como para inteiro, `double *q`, declara um ponteiro para um `double`;

### b) Operações com ponteiros:

`&` retorna o endereço de memória de uma variável (ex: `&x`)

- Ou posição de um vetor (ex: `&v[i]`);
- Ou campo de uma struct (ex: `&data.mes`);
- Podemos salvar o endereço em um ponteiro (ex: `p = &x;`);

`*` acessa o conteúdo no endereço indicado pelo ponteiro;

- `*p` onde `p` é um ponteiro;
- Podemos ler (ex: `x = *p;`) ou escrever (ex: `*p = 10;`);

Exemplo:

```
1 int *endereco;
2 int variavel = 90;
3 endereco = &variavel;
4 printf("Variavel: %d\n", variavel);
5 printf("Variavel: %d\n", *endereco);
6 printf("Endereço: %p\n", endereco);
7 printf("Endereço: %p\n", &variavel);
```



Os prints serão:

Variavel: 90

Variavel: 90  
Endereço: 127  
Endereço: 127

c) Vetores (Arrays) em C:

Em C, se fizermos `int v[100]` temos uma variável chamada `v`, que é, de fato, do tipo `int *`. `const`, `const` significa que não podemos fazer `v = &x`, isto é, não podemos mudar o endereço armazenado em `v`. Ele aponta para o primeiro `int` do vetor, ou seja, `v == &v[0];`

Para trabalhar com vetores, costumamos usar a biblioteca padrão do C `<stdlib.h>`, ela contém algumas funções importantes:

- **sizeof(int):**

Devolve o tamanho em bytes de um tipo de dado: `sizeof(int)` geralmente devolve 4 e caso fizermos de structures ele retornará a soma dos tamanhos dos seus membros e possivelmente mais alguns bytes para alinhamento.

- **malloc:**

Aloca dinamicamente a quantidade de bytes informada devolvendo o endereço da região de memória, a qual é sempre contígua. Exemplo:

-`malloc(sizeof(struct data))` aloca a quantidade de bytes necessária para representar uma `struct data`;

-`malloc(10 * sizeof(int))` aloca a quantidade de bytes necessária para representar 10 `ints`;

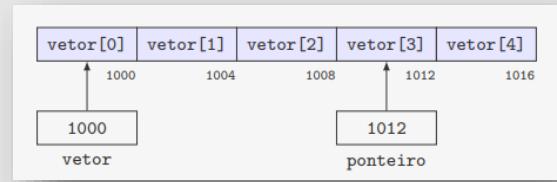
- **free:**

libera uma região de memória alocada dinamicamente, precisa ser um endereço que foi devolvido por `malloc` e evita que vazemos memória (memory leak);

d) Aritmética de ponteiros:

Podemos realizar operações aritméticas em ponteiros, como somar ou subtrair números inteiros, incrementar (`++`), decrementar (`--`), sendo que o compilador considera o tamanho do tipo apontado. Por exemplo somar 1 em um ponteiro para `int` faz com que o endereço pule `sizeof(int)` bytes.

1 `int vetor [5] = {1, 2, 3, 4, 5};`  
2 `int *ponteiro;`  
3 `ponteiro = vetor + 2;`



```
4 ponteiro++;
5 printf("%d %d %d", *vetor, *(ponteiro - 1), *ponteiro);
```

Se tivermos um ponteiro p, podemos escrever p[i], como se fosse um vetor. Isto é o mesmo que escrever \*(p+i);

Segue um exemplo de código usando as ideias apresentadas:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     double media, *notas;           // será usado como um vetor
6     int n;                         // A variável n é a quantidade de notas
7     scanf("%d", &n);
8     notas = malloc(n * sizeof(double));
9     if (notas == NULL) {
10         printf("Nao ha memoria suficiente!\n");
11         exit(1);                  // encerra o programa retornando 1
12     }
13     for (int i = 0; i < n; i++)
14         scanf("%lf", &notas[i]);
15     media = 0;
16     for (int i = 0; i < n; i++)
17         media += notas[i] / n;
18     printf("Média: %lf\n", media);
19     free(notas);
20     return 0;
21 }
```

A parte crucial para entender o código acontece na linha 8, nela:

-**sizeof(double)**: Calcula o tamanho em bytes de uma única variável double.

-**n \* sizeof(double)**: Multiplica esse tamanho pela quantidade n de notas, calculando o total de memória necessária.

-**malloc(...)**: Pede ao sistema operacional essa quantidade total de memória. Se conseguir, ele retorna o endereço do início do bloco de memória.

-**notas = ...** : O endereço retornado por malloc é armazenado no ponteiro notas. Agora, notas aponta para um espaço de memória grande o suficiente para guardar n valores do tipo double.

Assim o programador conseguiu pedir para o usuário um valor como “tamanho” para o vetor armazenou seu tamanho na memória e, só depois, baseando-se nisso criou o vetor;

Vale ressaltar o comando **free** da linha 19 ele é essencial e libera a memória que foi alocada com malloc. Isso “devolve” o bloco de memória ao sistema operacional para que possa ser usado por outros programas. Esquecer de usar free causa um “vazamento de memória” (*memory leak*).

e) Organização da memória:

A memória de um programa é dividida em duas partes:

- **Pilha:** onde são armazenadas as variáveis – Em geral, espaço limitado (ex: 8MB);
- **Heap:** onde são armazenados os outros dados – Do tamanho da memória RAM disponível;

Existem dois tipos de alocação:

- **estática(variáveis):** O compilador reserva um espaço na pilha, a variável é acessada por um nome bem definido e o espaço é liberado quando a função termina.
- **Dinâmica:** malloc reserva um número de bytes no heap, devemos guardar o endereço da variável com um ponteiro e ele deve ser liberado usando free;

Receita para alocação dinâmica de vetores:

- 1) Incluir a biblioteca **<stdlib.h>**;
- 2) Declare o ponteiro com o tipo apropriado, como **int \*v**;
- 3) Aloque a região de memória com malloc e o tamanho pode ser obtido com sizeof, como **v = malloc(n \* sizeof(int))**;
- 4) Verifique se acabou a memória comparando com NULL, use a função exit para sair do programa:

```
1 if (v == NULL) {  
2     printf("Nao ha memoria suficiente!\n");  
3     exit(1);  
4 }
```

- 5) Libere a memória após a utilização com free, como **free(v)**;

f) Ponteiros, vetores e funções:

Funções não podem devolver vetores, mas podem devolver ponteiros, podemos escrever: **int \* funcao(...);**

Nunca devolva o endereço de uma variável local, ela deixará de existir quando a função terminar, ou seja, nunca devolva o endereço de um vetor alocado estaticamente;

Exemplo:

- 1) Escreva uma função que dado um int n, aloca um vetor de double com n posições zeradas:

```

1 double * aloca_e_zera(int n) {
2     double *v = malloc(n * sizeof(double));
3     for (int i = 0; i < n; i++)
4         v[i] = 0.0;
5     return v;
6 }
```

Obs: Na mesma biblioteca <stdlib.h> tem a função calloc, que faz a mesma coisa dessa função:

```

1 double* aloca_e_zera_melhorado(int n) {
2     double* v = (double*) malloc(n, sizeof(double));
3     return v;
4}
```

2) Faça uma função que imprima vetores:

```

1 void imprime(double *v, int n) {
2     for (int i = 0; i < n; i++)
3         printf("%lf", v[i]);
4     printf("\n");
5 }
```

#### g) Ponteiros e structs:

Frequentemente alocamos uma struct dinamicamente elas serão o elemento básico de muitas Estruturas de Dados, teremos o ponteiro para uma struct e precisaremos acessar um dos seus campos.

## **6. Curso de C (Parte 6):**

### a) Passagem de parâmetros em Python:

<pre> 1 def adiciona(x): 2     x = x + 1 3 4 x = 10 5 adiciona(x) 6 print(x)</pre>	<pre> 1 def adiciona(lista): 2     lista.append(3) 3 4 lista = [1, 2] 5 adiciona(lista) 6 print(lista)</pre>	<pre> 1 def adiciona(lista): 2     lista = [1, 2, 3] 3 4 lista = [1, 2] 5 adiciona(lista) 6 print(lista)</pre>
--	--	--

Os três códigos exploram a diferença fundamental entre a passagem de tipos de dados imutáveis e mutáveis para funções em Python. O primeiro código imprime 10, pois **int** é um tipo imutável, a função adiciona cria uma cópia local da variável e a original permanece inalterada.

O segundo código imprime [1, 2, 3], porque listas são mutáveis e o método `.append()` modifica o objeto original diretamente, fazendo com que a alteração dentro da função seja refletida fora dela.

Já o terceiro código imprime [1, 2], pois, embora uma lista mutável seja passada, a operação de atribuição (`lista = [1, 2, 3]`) dentro da função apenas faz a variável local apontar para um objeto completamente novo, deixando a lista original, de fora da função, intacta.

b) Passagem de parâmetros em C:

Considere o seguinte código:

```
1 void imprime_invertido(int v[10], int n) {  
2     while (n > 0) {  
3         printf("%d ", v[n-1]);  
4         n--;  
5     }  
6 }  
7  
8 int main() {  
9     int n = 10, v[10] = {0,1,2,3,4,5,6,7,8,9};  
10    imprime_invertido(v, n);  
11    printf("%d\n", n);  
12    return 0;  
13 }
```

O valor da variável local `n` da função `main` é copiado para o parâmetro (variável local) `n` da função `imprime_invertido` (O valor de `n` em `main` não é alterado, continua 10).

Toda passagem de parâmetros em C é feita por cópia, o valor da variável (ou constante) da chamada da função é copiado para o parâmetro, mesmo se variável e parâmetro tiverem o mesmo nome. Mas e se passarmos um vetor?

```
1 void soma_um(int v[10], int n) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         v[i]++;  
5}
```

Quando passamos um vetor, na verdade, passamos o endereço do vetor por cópia, ou seja, o conteúdo do vetor não é passado para a função. Por isso mudanças dentro da função afetam o vetor. **Toda** passagem de parâmetros em C é feita por **cópia**.

Analisando novamente o primeiro código, mas com um ponteiro para n na função:

```
1 void imprime_invertido(int v[10], int *n) {  
2     while (*n > 0) {  
3         printf("%d ", v[*n-1]);  
4         *n--;  
5     }  
6 }  
7  
8 int main() {  
9     int n = 10, v[10] = {0,1,2,3,4,5,6,7,8,9};  
10    imprime_invertido(v, n);  
11    printf("%d\n", n);  
12    return 0;  
13 }
```

Agora, o resultado de n printado na linha 11 da main seria 0 e não mais 10, já que passamos um ponteiro para a função, ela sabe onde está na memória e, assim, consegue modificá-lo.

c) Alocação dinâmica:

No Python ele enviava a referência do objeto para a função e por isso podíamos modificá-lo se este fosse mutável, é algo parecido com o que C faz com vetores, a função sabe onde o objeto está a memória, isso pode ser bem mais rápido que copiar o objeto. É o que chamamos de passagem por referência e que no C é simulado com ponteiros, o Python faz a mesma coisa que o C, mas esconde do programador.

Se não estamos preocupados com eficiência de código, podemos ignorar a passagem por referência, os dois códigos seguintes são iguais, um usa passagem por referência e outro não.

<pre>1 void soma_um(int *x) { 2     *x = *x + 1; 3 } 4 5 int main() { 6     int y = 10; 7     soma_um(&amp;y); 8     printf("%d\n", y); 9     return 0; 10 }</pre>	<pre>1 int soma_um(int x) { 2     return x + 1; 3 } 4 5 int main() { 6     int y = 10; 7     y = soma_um(y); 8     printf("%d\n", y); 9     return 0; 10 }</pre>
--	--

d) Alocação dinâmica de matrizes:

Uma matriz alocada dinamicamente é como um vetor de vetores, matriz é um vetor alocado dinamicamente, cada `matriz[i]` é um vetor alocado dinamicamente e cada `matriz[i][j]` é o elemento na linha  $i$  e coluna  $j$ .

Um vetor de `int` alocado dinamicamente é do tipo `int *`, uma matriz é um vetor de vetores, porante seu tipo é `int**`(armazena um endereço de um ponteiro de `int`).

**7. Recursão:**

a) Ideias iniciais:

A ideia é que um problema pode ser resolvido em três etapas. Primeiramente, definimos as soluções para os casos básicos. Em seguida, tentamos reduzir o problema para instâncias menores do problema e, finalmente, combinamos o resultado das instâncias menores para obter um resultado do problema original. Como exemplo uma função que calcula factorial e outra que verifica se uma palavra é palíndroma:

```
1 int fat(int n) {
2     if (n == 0)                      // caso base
3         return 1;
4     else // caso geral
5         return n * fat(n - 1);      // instância menor
6 }
```

```
1 int eh_palindromo(char *palavra , int ini, int fim) {
2     if (ini >= fim)
3         return 1;
4     return palavra[ini] == palavra[fim] &&
5            eh_palindromo(palavra , ini + 1, fim - 1);
6 }
7
8 eh_palindromo (palavra , 0, strlen(palavra) - 1)
```

b) Busca binária:

Para buscar  $x$  no vetor ordenado “`dados`” entre as posições  $l$  e  $r$ .

Caso base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor;
- Se  $dados[m] == x$ , onde  $m = (l + r)/2 \rightarrow$  Devolvemos  $m$ ;

### Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$ ;
  - Devolvemos o resultado da chamada recursiva;

```
1 int busca_binaria(int *dados, int l, int r, int x) {  
2     int m = (l + r)/2;  
3     if (l > r)  
4         return -1;  
5     if (dados[m] == x)  
6         return m;  
7     else if (dados[m] < x)  
8         return busca_binaria(dados, m + 1, r, x);  
9     else  
10        return busca_binaria(dados, l, m - 1, x);  
11 }
```

### c) Recursão x algoritmos iterativos:

Normalmente algoritmos recursivos são mais simples de entender, por serem menores e mais fáceis de programar, além de serem muito mais elegantes. Algumas vezes podem ser MUITO ineficientes quando comparados a algoritmos iterativos.

```
1 int fib_rec(int n) {  
2     if (n == 1)  
3         return 1;  
4     else if (n == 2)  
5         return 1;  
6     else  
7         return fib_rec(n-2) +  
8             fib_rec(n-1);  
9 }  
  
1 int fib_iterativo(int n) {  
2     int ant, atual, prox, i;  
3     ant = atual = 1;  
4     for (i = 3; i <= n; i++) {  
5         prox = ant + atual;  
6         ant = atual;  
7         atual = prox;  
8     }  
9     return atual;  
10 }
```

Número de operações:

- iterativo:  $\approx n$ ;
- recursivo:  $\approx \text{fib}(n)$  (aproximadamente  $1.6^n$ );

## **8. Noções de Eficiência de Algoritmos:**

### a) Introdução e busca sequencial:

A execução de uma função/algoritmo depende de vários fatores, como o computador que ela foi rodada, quantos passos ele tem e sua complexidade. Para exemplificar utilizaremos um código simples de busca sequencial:

```

1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }

```

Como citado sua execução depende do computador que ela for rodada e de  $n$  (a quantidade de dados). Em geral, queremos analisar o pior caso do algoritmo, a análise do caso médio é mais difícil, normalmente probabilística.

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável);
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação) – No pior caso, essa linha é executada  $n + 1$  vezes;
- Linha 4: tempo  $c_4$  (acessos, comparação e if) – No pior caso, essa linha é executada  $n$  vezes;
- Linha 5: tempo  $c_5$  (acesso e return);
- Linha 6: tempo  $c_6$  (return);

Dessa forma, o tempo de execução é menor ou igual a:

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada uma dessas constantes depende exclusivamente da máquina a qual o código está sendo rodado, seja  $a = c_2 + c_3 + c_5 + c_6$ ,  $b = c_3 + c_4$  e  $d = a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

$$\begin{aligned} c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n = d \cdot n \end{aligned}$$

Isto é, o crescimento do tempo é linear em  $n$  (Cheio de carteação);

Como vimos, existe uma constante  $d$  tal que, para  $n \geq 1$ ,  $c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$ ,  $d$  não interessa tanto, depende apenas do computador (estamos preocupados em estimar). O tempo do algoritmo é da ordem de  $n$ , ou seja, a ordem de crescimento do tempo é igual a de  $f(n) = n$ . Dizemos que  $c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 = O(n)$ .

### b) Busca binária:

```

1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }

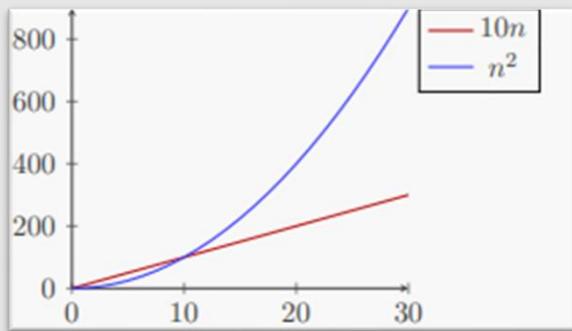
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ? Primeiro chamamos para  $n$ , depois para  $n/2$ ,  $n/4$ , ...,  $n/2^{t-1}$ . No pior caso só paramos quando  $n/2^t < 1 < n/2^{t-1}$ . Ou seja,  $t \leq 1 + \lg n$ , gastamos um tempo constante  $c$  em cada chamada. Para  $n \geq 1$ , o consumo de tempo é no máximo:

- $ct \leq c + c \lg n \leq 2c \lg n = O(\lg n)$ .

Temos dois objetivos para analisar algoritmo: Primeiro queremos entender o tempo de execução dele, exemplo a busca linear é  $O(n)$ , vamos dizer que ela é  $O(f(n))$ , assim, podemos compará-la com outro algoritmo que seja, por exemplo,  $O(\lg n)$ , que esse é melhor matematicamente que o primeiro.

Comparando funções  $10n$  e  $n^2$ :



Ao invés de comparar para todo  $n$  é útil comparar apenas para  $n$  suficientemente grandes, já que para  $n$  pequenos o programa roda quase instantaneamente.

c) Nomenclatura e consumo de tempo:

- **$O(1)$ :** tempo constante:
  - Não depende de  $n$ ;
  - Ex: atribuição e leitura de uma variável;
  - Ex: operações aritméticas:  $+$ ,  $-$ ,  $*$ ,  $/$ ;
  - Ex: comparações ( $\leq$ ,  $=$ ,  $\geq$ ,  $>$ ,  $\neq$ );
  - Ex: operadores booleanos ( $\&\&$ ,  $\&$ ,  $\|$ ,  $|$ ,  $!$ );
  - Ex: acesso a uma posição de um vetor;
- **$O(\lg n)$ :** logarítmico
  - $\lg$  indica  $\log_2$
  - Quando  $n$  dobra, o tempo aumenta em uma constante;
  - Ex: Busca binária;
  - Outros exemplos durante o curso;
- **$O(n)$ :** linear
  - Quando  $n$  dobra, o tempo dobra;
  - Ex: Busca linear;
  - Ex: Encontrar o máximo/mínimo de um vetor;
  - Ex: Produto interno de dois vetores;

- $O(n \lg n)$ :

- Quando n dobra, o tempo um pouco mais que dobra;
- Ex: algoritmos de ordenação que veremos;

- $O(n^2)$ : quadrático

- Quando n dobra, o tempo quadruplica;
- Ex: BubbleSort, SelectionSort e InsertionSort;

- $O(n^3)$ : cúbico

- Quando n dobra, o tempo octuplica;
- Ex: multiplicação de matrizes  $n \times n$ ;

## 9. Vetores:

Vetores em C são uma forma nativa da própria linguagem de estruturar dados, são listas indexadas de itens, blocos sequenciais de memória. Vetores podem ser alocados estaticamente ou dinamicamente. Sua grande vantagem é o acesso em tempo constante  $O(1)$  a qualquer um dos seus elementos através do índice.

### a) TAD vetor:

Struct vetor:

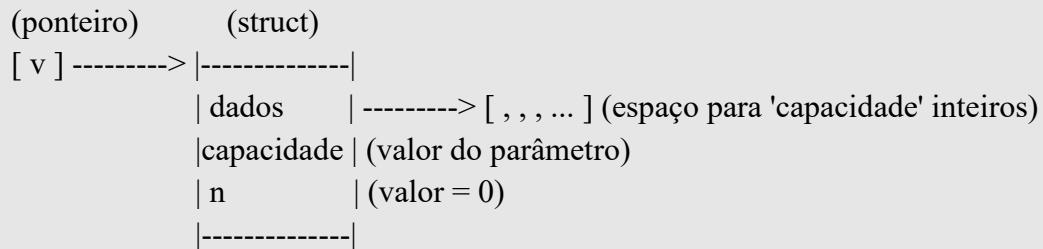
```
1 typedef struct vetor *p_vetor;
2
3 struct vetor { // para armazenar int
4     int *dados; // espaço na memória para armazenamento
5     int n; // número de elementos armazenados
6     int capacidade; // capacidade de armazenamento
7 };
```

Criaremos algumas funções para essa struct:

```
1 p_vetor criar_vetor_vazio(int capacidade) {
2     p_vetor v = malloc(sizeof(struct vetor));
3     v->dados = malloc(capacidade * sizeof(int));
4     v->capacidade = capacidade;
5     v->n = 0;
6     return v;
7 }
```

guarda na estrutura a informação de quantos elementos ela consegue armazenar. A linha 6 mostra que o vetor está vazio.

Essa função **cria o vetor**. A linha 2 cria um ponteiro para a estrutura. A linha 3 aloca a memória para o array interno que guardará números, guardado no campo “dados” da estrutura. A linha 4



```

1 void destruir_vetor(p_vetor v) {
2     free(v->dados);
3     free(v);
4 }

```

Essa função, libera a memória dos vetores que não serão mais usados, primeiro libera a memória dos dados e, posteriormente, da estrutura organizadora.

O código do cliente ficaria algo do tipo:

```

1 p_vetor v;
2 v = criar_vetor(100);
3 destruir_vetor(v);

```

Na linha 1 estamos declarando o ponteiro p\_vetor que é o “apelido” para struct vetor\*, um ponteiro para a struct.

```

1 void adicionar_elemento(p_vetor v, int valor) {
2     v->dados[v->n] = valor;
3     v->n++;
4 }

```

A função adiciona um elemento no final do vetor em O(1), sem se preocupar em manter nenhum

tipo de ordem. V->n é o truque dessa função, n guarda o número de elementos que já estão no vetor. Se o vetor tem 3 elementos, eles estão nos índices 0,1 e 2. O valor de v->n será 3, que é exatamente o índice da próxima posição livre.

Para criar uma função que remove elementos em tempo O(1), trocamos o elemento com o último v->dados[indice] = v->dados[v->n - 1]; e posteriormente diminuímos v->n: “v->n--”. Essa função de remover é muito eficiente, porém não preserva a ordem original dos elementos.

```

1 int busca(p_vetor v, int valor) {
2     for (int i = 0; i < v->n; i++)
3         if (v->dados[i] == valor)
4             return i;
5     return -1;
6 }

```

A função ao lado faz uma **busca sequencial** em  $O(n)$ , não podemos fazer uma busca binária, já que o vetor não está ordenado. A função retorna a posição do elemento se ele está no vetor e -1 caso contrário. Se as buscas no vetor forem muito frequentes no código é mais vantajoso ordenar o vetor e fazer uma busca binária, mesmo que ordenar a lista custe  $O(n^2)$  caso o algoritmo seja InsertionSort, SelectionSort ou BubbleSort.

A função adiciona elementos em um vetor ordenado, para isso primeiro encontra a posição correta, desloca os outros elementos para a direita e insere o elemento desejado na posição correta.

```

1 void adicionar_elemento(p_vetor v, int valor) {
2     int i;
3     for (i = v->n - 1; i >= 0 && v->dados[i] > valor; i--)
4         v->dados[i + 1] = v->dados[i];
5     v->dados[i + 1] = valor;
6     v->n++;
7 }

```

```

1 void remover_elemento_na_posicao(p_vetor v, int posicao) {
2     for(; posicao < v->n - 1; posicao++)
3         v->dados[posicao] = v->dados[posicao + 1];
4     v->n--;
5 }

```

A função que remove elementos de um vetor ordenado é muito parecida.

```

1 int busca_binaria(int *dados, int inicio, int fim, int valor) {
2     int meio = (inicio + fim) / 2;
3     if (inicio > fim)
4         return -1;
5     if (dados[meio] == valor)
6         return meio;
7     else if (dados[meio] < valor)
8         return busca_binaria(dados, meio + 1, fim, valor);
9     return busca_binaria(dados, inicio, meio - 1, valor);
10 }
11
12 int buscar(p_vetor v, int valor) {
13     return busca_binaria(v->dados, 0, v->n - 1, valor);
14 }

```

A função ao lado realiza uma **busca binária** em  $O(\lg n)$ , vale ressaltar que o cliente não precisa usar a busca binária diretamente, ele pode simplesmente chamar a função buscar, a busca\_binária não precisa nem estar no .h.

Vetores não ordenados tem inserção e remoção em  $O(1)$  e busca em  $O(n)$ , enquanto vetores ordenados tem inserção e remoção em  $O(n)$  e busca em  $O(\lg n)$ . Se temos muitas inserções e remoções e poucas buscas usamos vetores não ordenados, se temos o caso contrário usamos vetores ordenados.

Os vetores que vimos até agora tem um problema, eles têm espaço limitado, na hora de inicializar é necessário saber o tamanho máximo que o vetor terá durante seu tempo de vida, isso nem sempre é possível e pode gerar um grande desperdício de memória. Uma opção para isso é criar um vetor que aumenta e diminui de tamanho de acordo com a quantidade de dados armazenada.

```
1 void adicionar_elemento(p_vetor v, int valor) {
2     int *temp;
3     if (v->n == v->capacidade) {
4         temp = v->dados;
5         v->capacidade *= 2;
6         v->dados = malloc(v->capacidade * sizeof(int));
7         for (int i = 0; i < v->n; i++)
8             v->dados[i] = temp[i];
9         free(temp);
10    }
11    v->dados[v->n] = valor;
12    v->n++;
13 }
```

A função adicionar elemento adiciona um elemento no vetor dinamicamente, dobrando o tamanho do vetor se ao adicionar um elemento nele seu tamanho estoura. A linha 2 cria um ponteiro “temp” para guardar temporariamente o endereço dos dados antigos se

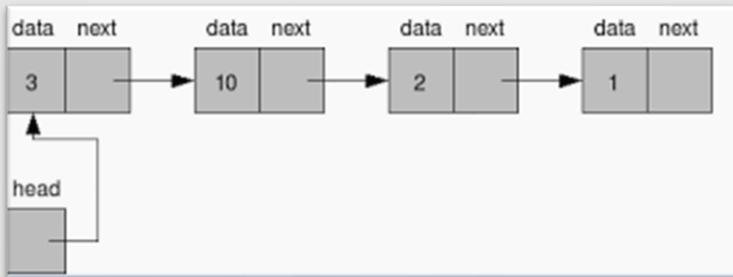
o vetor precisar crescer, a linha 3 verifica se o vetor está cheio e executa as seguintes passagens se estiver: Primeiro salva o endereço do array antigo(cheio) no ponteiro temp para não perder “temp = v->dados”, depois dobra a capacidade do vetor e aloca um bloco de memória novo e faz v -> dados apontar para ele, posteriormente é feito um loop que copia todos os elementos do array antigo(acessado via temp) para o novo e maior array, finalmente, a memória do array antigo é liberada. Após, com o espaço já garantido o elemento é adicionado. Esse código tem tempo O(1) se não precisou aumentar o valor e O(i) se precisou aumentar o valor para inserir o i-ésimo elemento.

## 10. Listas Ligadas:

### a) Introdução:

Vetores estão alocados contiguamente na memória, pode ser que tenhamos espaço nela, mas não para adicionar um vetor do tamanho desejado. Eles têm um tamanho fixo, ou alocamos um vetor pequeno e o espaço pode acabar ou alocamos um vetor grande e “desperdiçamos” memória. Para resolver isso vimos como alocá-los dinamicamente, contudo isso pode ser muito lento.

Uma alternativa para isso são as listas ligadas, alocamos memória conforme o necessário, guardamos um ponteiro para a estrutura em uma variável, o primeiro nó aponta para o segundo, o segundo aponta para o terceiro e o último nó aponta para NULL.



Nó: elemento alocado dinamicamente que contém um conjunto de dados, um ponteiro para outro nó.

Lista ligada: Conjunto de nós ligados entre si de maneira sequencial.

Obs: A lista ligada é acessada a partir de uma variável qualquer e um ponteiro pode estar “vazio” em C, isso significa que ele aponta para NULL.

b) Implementação:

Este código cria o modelo para um nó de lista ligada em C.

```
1 typedef struct no *p_no;
2
3 struct no {
4     int dado; // armazenamos int, mas poderia ser outro tipo
5     p_no prox;
6 };
```

A struct no contém um campo para guardar um dado (int dado) e um ponteiro (prox) que aponta para o próximo nó da lista, conectando um ao outro. A primeira linha, typedef, cria o apelido p\_no para simplificar a declaração de ponteiros para essa estrutura.

```
1 p_no criar_lista() {
2     return NULL;
3 }
```

A seguinte função retorna um ponteiro para o cabeçalho que aponta para lugar nenhum, ela está **criando uma lista vazia**. O código do cliente seria algo do tipo 1 p\_no lista; 2 lista = criar\_lista();

```
1 void destruir_lista(p_no lista) {
2     if (lista != NULL) {
3         destruir_lista(lista->prox);
4         free(lista);
5     }
6 }
```

A função destruir\_lista **apagando da memória a lista** chamada de forma recursiva.

```

1 p_no adicionar_elemento(p_no lista, int valor) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = valor;
5     novo->prox = lista;
6     return novo;
7 }

```

que é do tipo `p_no` (um ponteiro que será usado para guardar o endereço do novo nó que vamos criar). O segredo da função está na linha 5, o ponteiro `prox` do nosso novo nó é configurado para apontar para o antigo início da lista que a função recebeu como parâmetro, com isso, o novo nó “engata” na frente do que era o primeiro elemento. Na linha 6, como o novo nó foi adicionado no início ele passa a ser a nova “cabeça” da lista e, dessa forma, a função retorna o ponteiro novo para o começo da lista agora com um elemento a mais adicionado. O código do cliente ficaria algo do tipo: `1 lista = adicionar_elemento(lista, num);`

A inserção ocorre em  $O(1)$ , caso fosse adicionado no final na lista ocorreria em  $O(n)$ . Deveríamos verificar se `malloc` não devolve `NULL`, já que a memória poderia ter acabado.

```

1 void imprime(p_no lista) {
2     p_no atual;
3     for (atual = lista; atual != NULL; atual = atual->prox)
4         printf("%d\n", atual->dado);
5 }

```

A função `imprime`: **imprime interativamente os elementos da lista**. Em ordem reversa de inserção.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "lista_ligada.h"
4
5 int main() {
6     int num;
7     p_no lista;
8     lista = criar_lista();
9     // lê números positivos e armazena na lista
10    scanf("%d", &num);
11    while (num > 0) {
12        lista = adicionar_elemento(lista, num);
13        scanf("%d", &num);
14    }
15    imprime(lista); // em ordem reversa de inserção
16    destruir_lista(lista);
17    return 0;
18 }

```

Exemplo de um código na interface do cliente, que lê números, vais os adicionando na lista, depois os printa utilizando as funções acima, por fim libera toda a memória alocada e termina retornando 0.

```

1 p_no busca(p_no lista, int x) {
2     if (lista == NULL || lista->dado == x)
3         return lista;
4     return busca(lista->prox, x);
5 }

```

Função que **busca (recursivamente)** um elemento **x**, devolvendo o ponteiro para o nó encontrado ou NULL se o elemento não estiver na lista. A função retorna um ponteiro para o nó que possui o elemento “x”

procurado ou NULL se ele não estiver na lista.

```

3 p_no remove(p_no lista, int x) {
4     if (lista == NULL)
5         return NULL;
6     if (lista->dado == x) {
7         p_no proximo = lista->prox;
8         free(lista);
9         return proximo;
10    }
11    lista->prox = remove(lista->prox, x);
12    return lista;
13 }

```

A função **remove a primeira ocorrência** (caso exista) de um elemento **x** na lista ligada. Ela percorre a lista até encontrar o nó desejado. Ao encontrá-lo, a função o remove e retorna um ponteiro para o nó seguinte, permitindo que o nó anterior da cadeia se “religue” corretamente, pulando o elemento apagado. Se um nó não é o alvo, a função

continua a busca no restante da lista e se mantém no lugar, retornando o seu próprio endereço para preservar a estrutura.

```

3 p_no remove.todos(p_no lista, int x) {
4     if (lista == NULL)
5         return NULL;
6     lista->prox = remove.todos(lista->prox, x);
7     if (lista->dado == x) {
8         p_no proximo = lista->prox;
9         free(lista);
10        return proximo;
11    }
12    return lista;
13 }

```

A função **remove todas as ocorrências de um elemento x** de uma lista ligada dada. função remove todos apaga recursivamente todas as ocorrências de um valor **x**. Sua estratégia é ir até o final da lista primeiro e, então, na “volta” da recursão, ela verifica cada nó. Se um nó

contém o valor **x**, ele é removido e substituído pelo seu sucessor. Se não contém, ele permanece. Essa abordagem de analisar “de trás para frente” garante que múltiplos nós com o valor **x**, mesmo que consecutivos, sejam todos removidos corretamente.

c) Operações em listas ligadas:

```
1 p_no copiar_lista(p_no lista) {
2     p_no nova_lista = NULL, ultimo = NULL;
3     for (p_no p = lista; p != NULL; p = p->prox) {
4         p_no novo_no = malloc(sizeof(No));
5         novo_no->dado = p->dado;
6         novo_no->prox = NULL;
7         if (nova_lista == NULL)
8             nova_lista = novo_no;
9         else
10            ultimo->prox = novo_no;
11            ultimo = novo_no;
12    }
13    return nova_lista;
14 }
```

Função que **copia uma lista ligada**. A linha 2 cria o ponteiro que vai apontar para o primeiro elemento da nova lista, começando com NULL porque ela ainda está vazia, último vai apontar para o último nó da lista, ele é usado para adicionar novos nós no final da lista de forma mais eficiente, sem precisar percorrer ela inteira, posteriormente o loop cria um ponteiro que aponta para os elementos da lista original e os aloca na lista nova, depois copia dos dados para a lista nova. Após o laço terminar de percorrer toda a lista original, a função retorna o ponteiro nova\_lista, que é o endereço do primeiro nó da lista recém-criada, fechando a função.

```
1 p_no inverter_lista(p_no lista) {
2     p_no atual, ant, invertida = NULL;
3     atual = lista;
4     while (atual != NULL) {
5         ant = atual;
6         atual = atual->prox;
7         ant->prox = invertida;
8         invertida = ant;
9     }
10    return invertida;
11 }
```

Função que **inverte os elementos de uma lista ligada**. Ela percorre a lista original elemento por elemento. Para cada um, ela o remove do início da lista original e o adiciona no início de uma nova lista, chamada invertida. Ao final do processo, a lista invertida contém todos os elementos, mas na ordem reversa, e é retornada.

```
1 p_no concatenar_lista(p_no primeira, p_no segunda) {
2     if (primeira == NULL)
3         return segunda;
4     primeira->prox = concatenar_lista(primeira->prox, segunda);
5     return primeira;
6 }
```

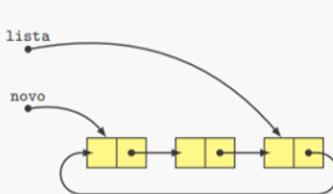
da primeira lista original e quando ela terminou retorna à segunda já com todos os elementos que tinham na primeira também. A linha 4 tem o passo recursivo, a função chama a si mesma, mas passando o resto da primeira lista(lista->prox). Ela atribui o resultado dessa chamada ao ponteiro prox do nó atual.

Função que **concatena duas listas ligadas**, A linha 2 tem o caso base da recursão, verifica se a primeira lista está vazia, isso ocorre quando chegamos ao final

d) Listas circulares:

```

1 p_no inserir_circular(p_no lista, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     if (lista == NULL) {
6         novo->prox = novo;
7         lista = novo;
8     } else {
9         novo->prox = lista->prox;
10        lista->prox = novo;
11    }
12    return lista;
13 }
```



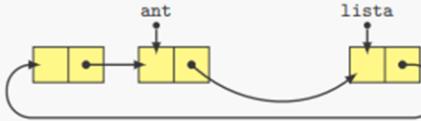
elemento (para formar um círculo, o seu ponteiro prox deve apontar para ele mesmo), o ponteiro principal da lista agora aponta para este único nó. Na linha 9 o ponteiro prox do novo nó é configurado para apontar para o que era o primeiro elemento da lista(lista ->prox). Assim, o novo nó agora aponta para o antigo início. Na linha 10 o ponteiro prox do último elemento (lista) é atualizado para apontar para o novo nó. Isso efetivamente insere o novo nó na lista, tornando-o o novo primeiro elemento.

Atenta-se que para o cliente a função para criar a lista é a mesma, o que vai diferenciar se a lista é circular ou ligada é a função de adicionar elemento.

A lista sempre aponta para o último elemento. O dado do primeiro nó é lista -> prox -> dado, o dado do último nó elemento é lista -> dado. Para inserir no final, basta devolver novo ao invés de lista.

```

1 p_no remover_circular(p_no lista, p_no no) {
2     p_no ant;
3     if (no->prox == no) {
4         free(no);
5         return NULL;
6     }
7     for(ant = no->prox; ant->prox != no; ant = ant->prox);
8     ant->prox = no->prox;
9     if (lista == no)
10        lista = ant;
11     free(no);
12     return lista;
13 }
```



linha 7, um laço for é usado para encontrar o nó anterior (ant) àquele que será removido. Na linha 8, a remoção acontece ao fazer o ponteiro prox do nó anterior (ant) 'pular' o nó a ser deletado, apontando diretamente para o seu sucessor (no->prox). A linha 9 checa se o nó removido era o último da lista e, se for, o ponteiro lista é atualizado na linha 10 para apontar para ant, que se torna o novo último elemento. Por fim, a memória do nó é liberada e a função retorna o ponteiro lista.

**A função insere em uma lista circular** Na linha 2 declaramos um ponteiro “novo” que apontará para o novo nó a ser criado, na linha 4 o valor x é atribuído ao campo dado do nó que acabou de ser criado. A linha 5 verifica se a lista está vazia, caso especial que precisa ser tratado separadamente, se isso acontece o novo será o único

**A função remove um nó específico de uma lista circular.** A linha 3 verifica se o nó é o único elemento da lista (se ele aponta para si mesmo), um caso especial que é tratado liberando a memória do nó e retornando NULL, pois a lista ficará vazia. Caso contrário, na

```

1 void imprimir_lista_circular(p_no lista) {
2     p_no atual = lista->prox;
3     do {
4         printf("%d\n", atual->dado);
5         atual = atual->prox;
6     } while (atual != lista->prox);
7 }

```

A função percorre e imprime todos os elementos de uma lista circular. Na linha 2, um ponteiro atual é inicializado para apontar para o início da lista (lista->prox). A função usa um laço do-while, que é ideal para listas circulares pois executa o bloco ao menos uma vez antes de testar a condição. Dentro do laço, na linha 4 o valor do nó atual é impresso e na linha 5 o ponteiro avança para o próximo. O laço continua até que atual complete a volta e aponte novamente para o início da lista, garantindo que todos os elementos sejam impressos uma única vez. Atente-se que a função não pode ser usada se a lista fosse vazia, isso causaria um erro segmentation fault, o problema estaria em “p\_no atual = lista->prox;” caso a lista estivesse vazia, o ponteiro lista será NULL. Tentar acessar lista ->prox é o mesmo que tentar acessar um campo de um endereço de memória nulo, o que faz o programa dar erro. Podemos corrigir isso adicionando um if no começo da função do tipo :

```

if (lista == NULL) {
    return;
}

```

```

1 p_no concatena(p_no lista1, p_no lista2) {
2     p_no p = lista1;
3     while (p->prox != lista1)
4         p = p->prox;
5     p->prox = lista2;
6     p = lista2;
7     while (p->prox != lista2)
8         p = p->prox;
9     p->prox = lista1;
10    return lista1;
11 }

```

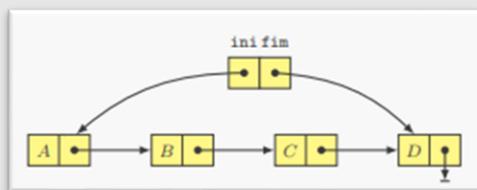
A função concatena une duas listas circulares em uma única lista circular maior. Diferente dos exemplos anteriores, este código assume que os ponteiros lista1 e lista2 apontam para o início (cabeça) de suas respectivas listas. Primeiramente, o laço while na linha 3 percorre a lista1 até encontrar seu último nó. Na linha 5, o ponteiro prox deste último nó é modificado para apontar para o início da lista2, efetivamente conectando o fim da primeira com o começo da segunda. Em seguida, o laço na linha 7 encontra o último nó da lista2 e, na linha 9, seu ponteiro prox é ajustado para apontar de volta para o início da lista1, fechando o novo e maior círculo. Por fim, a função retorna lista1, que é o início da nova lista combinada.

## **11. Filas:**

Para exemplificar uma fila utilizaremos o exemplo de uma impressora compartilhada por alunos de um laboratório, alunos enviam documentos ao mesmo tempo, gerenciaremos a lista de tarefas de impressão com uma fila.

Filas removem primeiro objetos inseridos há mais tempo, estruturas assim são chamadas de estruturas FIFO (first-in first-out), podemos adicionar as operações: enfileirar(queue) que adiciona itens no “fim” e desenfileirar (dequeue) que remove itens do “início”.

### a) Implementaremos com listas ligadas:



```
1 typedef struct fila *p_fila;
2
3 struct fila {
4     p_no ini, fim; // inicio e fim da fila
5 };
```

Struct para os ponteiros dos nós, início e fim da fila.

Códigos que criam e destroem filas:

```
1 p_fila criar_fila() {
2     p_fila f;
3     f = malloc(sizeof(struct fila));
4     f->ini = NULL;
5     f->fim = NULL;
6     return f;
7 }
```

```
1 void destruir_fila(p_fila f) {
2     destruir_lista(f->ini);
3     free(f);
4 }
```

```
1 void enfileira(p_fila f, int x) {
2     p_no novo;
3     novo = malloc(sizeof(struct no));
4     novo->dado = x;
5     novo->prox = NULL;
6     if (f->ini == NULL)
7         f->ini = novo;
8     else
9         f->fim->prox = novo;
10    f->fim = novo;
11 }
```

A função **adiciona um novo elemento ao final de uma fila**, que é implementada com uma lista ligada. Primeiramente, nas linhas 3 a 5, um novo nó é alocado na memória, o valor x é atribuído ao seu dado e seu ponteiro prox é definido como NULL, pois ele será o novo último da fila. A linha 6 verifica se a fila está vazia; se sim,

o ponteiro de início (f -> ini) também passa a apontar para este novo nó. Caso contrário, a linha 9 conecta o novo nó ao fim da lista, fazendo o prox do antigo último elemento (f->fim) apontar para ele. Finalmente, a linha 10 atualiza o ponteiro de fim da fila (f->fim) para que ele aponte para o novo nó, completando a operação.

```

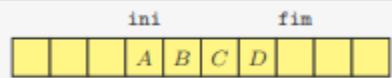
1 int desenfileira(p_fila f) {
2     p_no primeiro = f->ini;
3     int dado = primeiro->dado;
4     f->ini = f->ini->prox;
5     if (f->ini == NULL)
6         f->fim = NULL;
7     free(primeiro);
8     return dado;
9 }

```

A função **remove** e retorna o elemento que está no **início de uma fila**. Inicialmente, na linha 2, um ponteiro **primeiro** guarda o endereço do nó inicial e, na linha 3, o valor contido nesse nó é salvo na variável **dado**. A linha 4 efetivamente remove o elemento da fila, fazendo com que o ponteiro de início (**f->ini**) avance para o próximo nó da lista. A linha 5 trata o caso especial em que a fila fica vazia após a remoção; se isso acontecer, a linha 6 atualiza também o ponteiro de fim da fila (**f->fim**) para **NULL**. Por fim, a memória que era ocupada pelo antigo primeiro nó é liberada, e a função retorna o valor **dado** que foi extraído.

#### b) Implementando com vetores:

Primeira ideia, inserirmos no final do vetor: O(1) e removemos do começo do vetor O(n), ou temos uma variável **ini** indicando o começo da fila e uma **fim** indicando o seu fim.



E se, ao inserir, tivermos apenas espaço à esquerda de **ini**? Podemos mover toda a fila para o começo do vetor, contudo isso levaria tempo O(n).

Para solucionar isso podemos considerar o vetor de tamanho N circularmente, as manipulações de índices são realizadas módulo N.

```

1 typedef struct fila *p_fila;
2
3 struct fila {
4     int *v;
5     int ini, fim, N, tamanho;
6 };

```

Este código define a struct para uma fila baseada em vetor. Ele agrupa o ponteiro para o vetor de dados (**v**) junto com as variáveis de controle essenciais: os índices de início e fim (**ini, fim**) e os contadores de capacidade e tamanho atual (**N, tamanho**).

```

1 p_fila criar_fila(int N) {
2     p_fila f;
3     f = malloc(sizeof(struct fila));
4     f->v = malloc(N * sizeof(int));
5     f->ini = 0;
6     f->fim = 0;
7     f->N = N;
8     f->tamanho = 0;
9     return f;
10 }

```

A função aloca dinamicamente a memória e inicializa uma nova fila vazia com uma capacidade máxima N. Primeiramente, na linha 3, ela aloca a memória para a própria estrutura da fila e, na linha 4, aloca o espaço para o vetor interno que guardará os elementos. Em seguida, as linhas 5 a 8 inicializam as variáveis de controle: os índices de início e fim (ini e fim) começam em 0, a capacidade N é armazenada na estrutura, e o tamanho atual (tamanho) é definido como 0 para indicar que a fila começa vazia. Ao final, a função retorna o ponteiro para a estrutura da fila, já pronta para ser utilizada.

A função remove e retorna o elemento do início de uma fila implementada com um vetor circular. Na linha 2, o valor do primeiro elemento, localizado no índice f->ini do vetor, é primeiramente guardado na variável valor. A linha 3 é a chave da lógica circular, pois ela avança o índice de início (ini) para a próxima posição, usando o operador de módulo (% f->N) para garantir que o índice "dê a volta" para 0 caso chegue ao final do vetor. Em seguida, na linha 4, o tamanho da fila é decrementado e, por fim, a função retorna o valor que foi retirado da fila.

```

1 int desenfileira(p_fila f) {
2     int valor = f->v[f->ini];
3     f->ini = (f->ini + 1) % f->N;
4     f->tamanho--;
5     return valor;
6 }

```

```

1 void enfileira(p_fila f, int valor) {
2     f->v[f->fim] = valor;
3     f->fim = (f->fim + 1) % f->N;
4     f->tamanho++;
5 }

```

A função adiciona um novo elemento ao final de uma fila que utiliza um vetor circular. Na linha 2, o valor é inserido no vetor, na posição livre indicada pelo índice f->fim. A linha 3, então, avança o índice de fim (fim) para a próxima posição, usando o operador de módulo (% f->N) para assegurar que, ao chegar ao final do vetor, o índice "dê a volta" e retorne a 0. Por fim, na linha 4, o tamanho da fila é incrementado para refletir a adição do novo elemento.

Filas são aplicadas em gerenciamento de filas de impressão, buffer do teclado, escalonamento de processos, comunicação entre aplicativos/computadores, percurso de estrutura de dados complexas (grafos, etc...).

## **12. Pilhas:**

Pilhas removem primeiro objetos inseridos há menos tempo, são estruturas chamadas de LIFO (last-in first-out). É como uma pilha de pratos: empilha (push) os pratos limpos que já estão na pilha e desempilha (pop) o prato de cima para usar.

### **a) Implementação com vetores:**

Definimos a struct pilha baseada em um array. A struct contém um ponteiro \*v, que apontará para o vetor onde os dados serão efetivamente guardados, e um inteiro topo, o qual marca a posição do elemento que está no topo da pilha dentro do vetor v.

```
1 void empilar(p_pilha p, int valor) {  
2     p->v[p->topo] = valor;  
3     p->topo++;  
4 }
```

Definição:

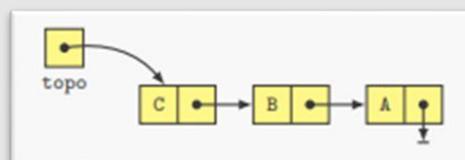
```
1 typedef struct pilha *p_pilha;  
2  
3 struct pilha {  
4     int *v;  
5     int topo;  
6 };
```

A função é responsável por **adicionar um novo elemento** ao topo da pilha. Primeiramente, ela insere o valor recebido na posição do vetor que é indicada pelo índice topo, que representa o primeiro espaço livre. Em seguida, ela incrementa o índice topo em uma unidade, fazendo com que ele passe a apontar para a próxima posição vaga, deixando a pilha pronta para uma futura inserção.

A função **remove** e retorna o elemento que está no topo da **pilha**. Para isso, ela primeiro decrementa em uma unidade o índice topo, fazendo com que ele, que antes apontava para um espaço livre, passe a apontar para o último elemento válido da pilha. Logo após, a função retorna o valor que se encontra nesta posição do vetor, completando a remoção.

```
1 int desempilar(p_pilha p) {  
2     p->topo--;  
3     return p->v[p->topo];  
4 }
```

### **b) Implementação com listas ligadas:**



```

1 typedef struct pilha *p_pilha;
2
3 struct pilha {
4     p_no topo;
5 };

```

Definindo a struct “pilha”, ela contém apenas um membro, o ponteiro “topo”, este aqui é um ponteiro do tipo `p_no` que aponta diretamente para o nó que está no topo da pilha, servindo de entrada para todos os outros elementos.

A função **adiciona um novo elemento no topo de uma pilha implementada com lista ligada**.

Primeiramente, um novo nó é alocado na memória na linha 2 e o valor é inserido nele na linha 3. A lógica principal acontece em seguida: a linha 4 faz o ponteiro prox do novo nó apontar para o que era o topo antigo da pilha, conectando-o ao restante da lista. Finalmente, a linha 5 atualiza o ponteiro topo da pilha para que ele aponte para este nó recém-criado, que passa a ser o novo elemento no topo.

```

1 void empilhar(p_pilha pilha, int valor) {
2     p_no novo = malloc(sizeof(struct no));
3     novo->dado = valor;
4     novo->prox = pilha->topo;
5     pilha->topo = novo;
6 }

```

```

1 int desempilhar(p_pilha pilha) {
2     p_no topo = pilha->topo;
3     int valor = topo->dado;
4     pilha->topo = pilha->topo->prox;
5     free(topo);
6     return valor;
7 }

```

A função **remove e retorna o elemento do topo de uma pilha de lista ligada**. Inicialmente, ela cria um ponteiro temporário para o nó do topo na linha 2 e, na linha 3, salva o valor contido nele em uma variável. A remoção em si ocorre na linha 4, onde o ponteiro topo da pilha é atualizado para apontar para o nó seguinte,

que se torna o novo topo. Com o nó antigo já desconectado da pilha, a linha 5 libera a sua memória para evitar vazamentos, e a função finaliza na linha 6 retornando o valor que foi salvo.

### c) Deque:

Um **deque (double ended queue)** é uma estrutura de dados com as operações de inserir e remover elementos no início e no fim. Implementaremos ele aqui utilizando listas ligadas.

A base da nossa implementação é uma lista duplamente ligada. Cada nó da lista (struct `no`) armazena não apenas um ponteiro para o próximo elemento (`prox`), mas também um ponteiro para o anterior (`ant`), o que permite a navegação eficiente nos dois sentidos. A estrutura principal `Deque` atua como um controlador, guardando apenas dois ponteiros: um para o início e outro para o fim da lista. Isso garante que o acesso a

```

1 typedef struct no {
2     int dado;
3     struct no *prox, *ant;
4 } No;
5
6 typedef No * p_no;
7
8 typedef struct {
9     p_no inicio, fim;
10 } Deque;
11
12 typedef Deque * p_deque;
13
14 p_deque cria_deque() {
15     p_deque d = malloc(sizeof(Deque));
16     d->inicio = d->fim = NULL;
17     return d;
18 }
19
20 void insere_inicio(p_deque d, int dado) {
21     p_no novo = malloc(sizeof(No));
22     novo->dado = dado;
23     novo->prox = d->inicio;
24     novo->ant = NULL;
25     if (d->inicio != NULL)
26         d->inicio->ant = novo;
27     else
28         d->fim = novo;
29     d->inicio = novo;
30 }

```

```

31 void insere_fim(p_deque d, int dado) {
32     p_no novo = malloc(sizeof(No));
33     novo->dado = dado;
34     novo->prox = NULL;
35     novo->ant = d->fim;
36     if (d->fim != NULL)
37         d->fim->prox = novo;
38     else
39         d->inicio = novo;
40     d->fim = novo;
41 }
42
43 int remove_inicio(p_deque d) {
44     p_no aux = d->inicio;
45     int dado = aux->dado;
46     d->inicio = d->inicio->prox;
47     if (d->inicio != NULL)
48         d->inicio->ant = NULL;
49     else
50         d->fim = NULL;
51     free(aux);
52     return dado;
53 }
54
55 int remove_fim(p_deque d) {
56     p_no aux = d->fim;
57     int dado = aux->dado;
58     d->fim = d->fim->ant;
59     if (d->fim != NULL)
60         d->fim->prox = NULL;
61     else
62         d->inicio = NULL;
63     free(aux);
64     return dado;
65 }

```

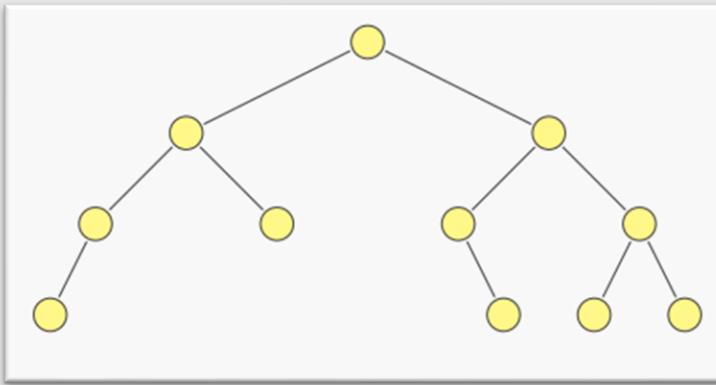
ambas as extremidades sejam imediatas, tornando as operações muito rápidas. A função `cria_deque` simplesmente aloca essa estrutura controladora e a inicializa como vazia, pronta para receber elementos.

As funções de inserção (`insere_inicio` e `insere_fim`) criam um nó e realizam o ajuste cuidadoso dos ponteiros. Ao adicionar um elemento, é crucial não só ligá-lo à lista, mas também atualizar o ponteiro `ant` do antigo primeiro nó ou o `prox` do antigo último nó para que apontem de volta ao novo elemento, mantendo a lista coesa. De forma simétrica, as funções de remoção (`remove_inicio` e `remove_fim`) desvinculam o nó da extremidade, ajustando o ponteiro `inicio` ou `fim` e garantindo que o novo nó da ponta tenha seu respectivo ponteiro `ant` ou `prox` ajustado para `NULL`. Todas as quatro operações tratam corretamente os casos especiais, como

operar em uma lista que está vazia ou que se torna vazia após a remoção.

### 13. Árvores Binárias:

**Árvores binárias** são estruturas hierárquicas, na qual cada elemento/nó, possui no máximo dois filhos, um a esquerda e um a direita. Seus elementos básicos são: os nós, ele armazena um ponteiro para os filhos, a raiz, que é o nó do topo da árvore ele não possui um “pai” sendo o ponto de partida da estrutura e subárvore, são árvores formadas por um nó e todos seus descendentes e suas folhas, que são os nós que não possuem filhos. Em uma árvore cada nó é a raiz de uma subárvore esquerda e uma subárvore direita.



Exemplo de uma árvore binária, ela pode ser ou o conjunto vazio ou um nó conectado a duas árvores binárias.

Define-se altura da árvore como o comprimento do caminho mais longo da raiz até uma folha, a altura de uma árvore com um único nó é 0. Se a altura da árvore é  $h$ , então a árvore tem no

mínimo  $h$  nós e tem no máximo  $2^h - 1$  nós. Se a árvore tem  $n \geq 1$  nós, então: a altura é no mínimo  $\lceil \lg(n+1) \rceil$  quando a árvore é completa e no máximo  $n$  quando cada nó não-terminal tem apenas um filho.

a) Implementação:

```

1 typedef struct no *p_no;
2
3 struct no {
4     int dado;
5     p_no esq, dir; /* pai */
6 };

```

typedef para struct no \*.

O código define a struct de um nó para uma árvore binária, contendo um campo de dados inteiro (dado) e dois ponteiros (esq e dir) que apontam para os nós filhos esquerdo e direito, respectivamente. Adicionalmente, ele cria um

A função `criar_arvore` serve para criar um único nó de uma árvore binária recursivamente. Ela aloca dinamicamente a memória necessária para este nó, atribui a ele um valor inteiro

```

1 p_no criar_arvore(int x, p_no esq, p_no dir) {
2     p_no r = malloc(sizeof(struct no));
3     r->dado = x;
4     r->esq = esq;
5     r->dir = dir;
6     return r;
7 }

```

(x) e conecta seus ponteiros esquerdo (esq) e direito (dir) às subárvore correspondentes, que são recebidas como parâmetros. Ao final, a função retorna o endereço de memória do nó recém-criado e configurado, pronto para ser integrado a uma estrutura de árvore maior.

```

1 p_no procurar_no(p_no raiz, int x) {
2     if (raiz == NULL || raiz->dado == x)
3         return raiz;
4     p_no esq = procurar_no(raiz->esq, x);
5     if (esq != NULL)
6         return esq;
7     return procurar_no(raiz->dir, x);
8 }

```

A função `procurar_no` implementa um algoritmo de busca recursiva para encontrar um nó com um valor  $x$  em uma árvore binária. A estratégia de busca segue a ordem de percurso pré-ordem

(raiz, esquerda, direita): primeiro, ela verifica se o nó atual contém o valor ou se é nulo; caso contrário, ela busca recursivamente em toda a subárvore esquerda; e somente se não encontrar nada na esquerda, ela busca em toda a subárvore direita. A função retorna um ponteiro para o primeiro nó encontrado com o valor correspondente ou retorna NULL se o valor não estiver presente na árvore.

```

1 int numero_nos(p_no raiz) {
2     if (raiz == NULL)
3         return 0;
4     return numero_nos(raiz->esq) + numero_nos(raiz->dir) + 1;
5 }

```

A função `numero_nos` calcula o número total de nós em uma árvore

binária recursivamente. Sua lógica se baseia na definição de que o tamanho de uma árvore é a soma dos tamanhos de suas subárvore direita e esquerda. O caso base da recursão acontece quando a árvore é vazia, que corretamente retorna o valor 0. Ao aplicar essa regra repetidamente, a função soma todos os nós da árvore e retorna a contagem total como um número inteiro.

A função `altura` calcula a altura de uma árvore binária. Ela opera com base no princípio de que a altura de qualquer árvore é 1 (contando o nó raiz) somado à altura da sua subárvore mais alta. Para isso, a função calcula recursivamente a altura das subárvore esquerda e direita, compara os dois valores para descobrir qual é o maior e retorna esse valor máximo somado a um. O caso base que finaliza a recursão é uma árvore vazia (um ponteiro `NULL`), para a qual a altura é retornada como 0.

```

1 int altura(p_no raiz) {
2     if (raiz == NULL)
3         return 0;
4     int h_esq = altura(raiz->esq);
5     int h_dir = altura(raiz->dir);
6     return 1 + (h_esq > h_dir ? h_esq : h_dir);
7 }

```

### Operadores condicionais ternários:

A sintaxe geral é a seguinte:

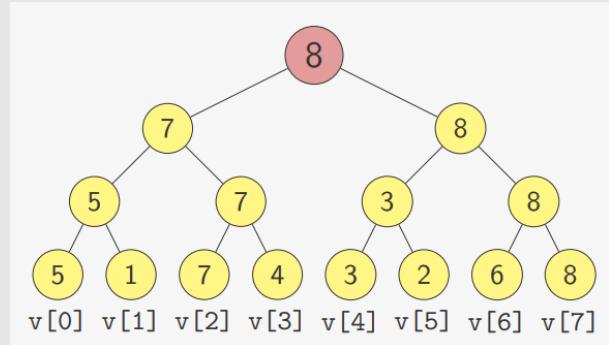
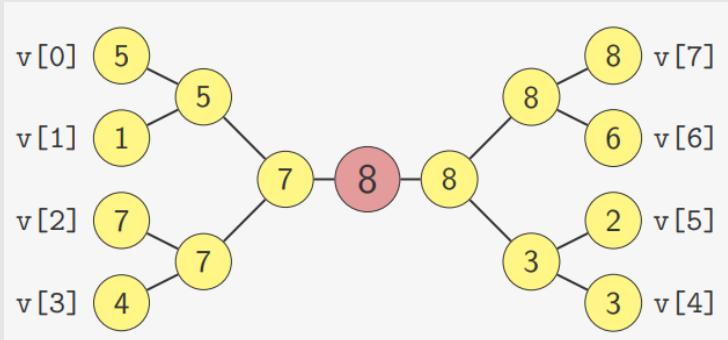
Condição ? valor se verdadeiro : valor se falso

A lógica é:

1. **condição:** O programa primeiro avalia a expressão antes do `?`.
2. `?:` Se a condição for **verdadeira**, a expressão inteira resulta no `valor_se_verdadeiro` (a parte entre o `?` e o `:`).
3. `::` Se a condição for **falsa**, a expressão inteira resulta no `valor_se_falso` (a parte depois do `:`).

### b) Criando um torneio:

Dado um vetor  $v$  com  $n$  números, queremos criar um torneio, decidindo qual é o maior número em um esquema de chaves, como o exemplo:



É uma **árvore binária**, onde o valor do pai é o maior valor dos seus filhos. Para resolver o torneio, resolveremos o torneio das duas subárvores recursivamente e decidiremos o vencedor.

```
1 p_no torneio(int *v, int inicio, int fim) {
2     if (inicio == fim)
3         return criar_arvore(v[inicio], NULL, NULL);
4     int meio = (inicio + fim) / 2;
5     p_no esq = torneio(v, inicio, meio);
6     p_no dir = torneio(v, meio + 1, fim);
7     int valor = esq->dado > dir->dado ? esq->dado : dir->dado;
8     return criar_arvore(valor, esq, dir);
9 }
```

A função **torneio** implementa um algoritmo recursivo para construir uma árvore de torneio a partir de um vetor. Ela divide repetidamente o vetor ao meio até chegar a

elementos individuais, que se tornam as folhas da árvore. Em seguida, na volta da recursão, ela "combina" as subárvores criando nós pais cujo valor é o máximo entre os valores de seus dois filhos. O resultado é uma árvore binária onde o valor de qualquer nó pai é sempre o maior valor contido em suas subárvores, e a raiz da árvore contém o maior valor de todo o vetor original.

### c) Percorrendo os nós:

Podemos percorrer os nós de uma árvore binária de três modos diferentes de profundidade:

1. Pré-ordem: Primeiro processa a raiz, depois a subárvore esquerda e por último a subárvore direita.
2. Pós-ordem: Primeiro processa a subárvore esquerda, depois a subárvore direita e por fim a raiz.
3. Inordem: Primeira visita a subárvore esquerda, depois a raiz e por último a subárvore direita.

```

1 void pre_ordem(p_no raiz) {
2     if (raiz != NULL) {
3         printf("%d ", raiz->dado);
4         pre_ordem(raiz->esq);
5         pre_ordem(raiz->dir);
6     }
7 }
```

percorre recursivamente toda a subárvore esquerda; e terceiro, após a esquerda estar completa, ele percorre recursivamente toda a subárvore direita. Isso resulta em uma impressão dos nós na ordem exata: Raiz, Esquerda, Direita.

A função `pos_ordem` realiza um percurso em pós-ordem. Para qualquer nó, o algoritmo segue a sequência estrita de primeiro percorrer recursivamente toda a subárvore esquerda, depois percorrer recursivamente toda a subárvore direita e,

somente ao final, processar o dado do próprio nó (neste caso, imprimindo-o). O resultado é uma visita a todos os nós no padrão Esquerda-Direita-Raiz, o que é útil em operações como a liberação de memória da árvore, pois garante que os filhos sejam processados antes do pai.

```

1 void inordem(p_no raiz) {
2     if (raiz != NULL) {
3         inordem(raiz->esq);
4         printf("%d ", raiz->dado);
5         inordem(raiz->dir);
6     }
7 }
```

recursivamente toda a subárvore direita.

Também podemos realizar a busca em largura, visitando os nós por níveis, da esquerda para a direita. A função `percurso_em_largura` implementa um percurso em nível de forma iterativa, utilizando uma fila como estrutura de dados auxiliar. O algoritmo começa inserindo a raiz da árvore na fila. Em seguida, ele entra em um laço que se repete enquanto a fila não estiver

A função `pre_ordem` implementa o percurso em pré-ordem de uma árvore binária de forma recursiva. Para cada nó da árvore, o algoritmo segue estritamente três passos: primeiro, ele processa o dado do nó atual (neste caso, imprimindo-o na tela); segundo ele

```

1 void pos_ordem(p_no raiz) {
2     if (raiz != NULL) {
3         pos_ordem(raiz->esq);
4         pos_ordem(raiz->dir);
5         printf("%d ", raiz->dado);
6     }
7 }
```

A função `inordem` implementa o percurso em-ordem. Para qualquer nó, o algoritmo segue a ordem: primeiro, percorre recursivamente toda a subárvore esquerda; em segundo lugar, processa o dado do nó atual (imprimindo-o); e, por último, percorre

```

1 void percurso_em_largura(p_no raiz) {
2     p_fila fila = criar_fila();
3     enfileirar(fila, raiz);
4     while(!fila_vazia(fila)) {
5         raiz = desenfileirar(fila);
6         if (raiz != NULL) {
7             enfileirar(fila, raiz->esq);
8             enfileirar(fila, raiz->dir);
9             printf("%d ", raiz->dado); /* v
10        */
11    }
12    destruir_fila(fila);
13 }
```

vazia, a cada passo removendo o primeiro nó da fila, imprimindo seu valor e, em seguida, adicionando seus filhos (esquerdo e direito, se existirem) ao final da fila. Essa abordagem de "primeiro a entrar, primeiro a sair" garante que os nós sejam visitados nível por nível, explorando completamente a largura da árvore em uma profundidade antes de passar para a próxima.

d) Funções úteis:

```
1 int folhas(p_no raiz) {
2     if (raiz == NULL)
3         return 0;
4     if (raiz->esq == NULL && raiz->dir == NULL)
5         return 1;
6     return folhas(raiz->esq) + folhas(raiz->dir);
7 }
```

nulo) não tem folhas (retorna 0), e um nó que não possui filhos à esquerda nem à direita é uma folha (retorna 1). Para qualquer outro nó interno, a função calcula o total de folhas somando o resultado da contagem de folhas em sua subárvore esquerda com o resultado da contagem em sua subárvore direita. Ao aplicar essa regra repetidamente, a função percorre a árvore e acumula a contagem de todos os nós que satisfazem a definição de folha.

A função `apaga_folhas` remove de uma árvore binária todas as folhas que contêm um valor específico x. Ela opera de forma recursiva, utilizando um percurso do tipo pós-ordem, ou seja, processando os filhos antes do pai. A função desce até as folhas da árvore e, se uma folha com o valor x é encontrada, seu pai, que então atualiza seu ponteiro para todos os outros nós que não são apagados, mantendo a estrutura da árvore intacta. A modificada.

```
1 int iguais(p_no arvore1, p_no arvore2) {
2     if (arvore1 == NULL && arvore2 == NULL)
3         return 1;
4     if (arvore1 == NULL || arvore2 == NULL)
5         return 0;
6     return arvore1->dado == arvore2->dado &&
7             iguais(arvore1->esq, arvore2->esq) &&
8             iguais(arvore1->dir, arvore2->dir);
9 }
```

A função `folhas` é uma função recursiva que conta o número de nós folha em uma árvore binária. Sua lógica baseia-se em dois casos de parada: uma árvore vazia (`nó`

nos à esquerda nem à direita é uma

```
1 p_no apaga_folhas(p_no raiz, int x) {
2     if (raiz == NULL)
3         return NULL;
4     if (raiz->esq == NULL && raiz->dir == NULL &&
5         raiz->dado == x) {
6         free(raiz);
7         return NULL;
8     }
9     raiz->esq = apaga_folhas(raiz->esq, x);
10    raiz->dir = apaga_folhas(raiz->dir, x);
11    return raiz;
12 }
```

A função `iguais` compara recursivamente duas árvores binárias para determinar se elas são estruturalmente idênticas e se os nós correspondentes contêm os mesmos valores. O algoritmo funciona

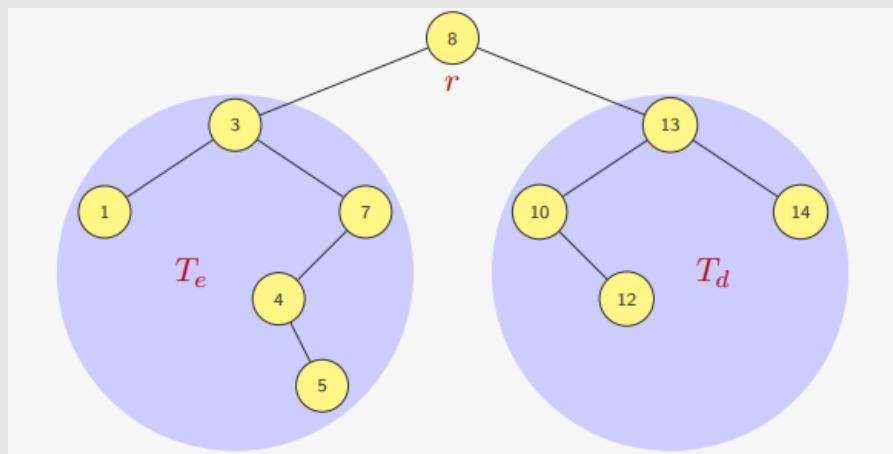
atravessando ambas as árvores simultaneamente. Ele retorna verdadeiro (1) se ambas as árvores estiverem vazias ou se, para cada nó, o valor for o mesmo em ambas as árvores e, recursivamente, suas subárvore esquerdas e direitas também forem idênticas. A função retorna falso (0) assim que encontra a primeira discrepância, seja uma diferença nos valores dos dados ou uma diferença na estrutura das árvores.

#### **14. Árvores Binárias de busca:**

Usando listas duplamente ligadas podemos inserir e remover em  $O(1)$ , mas a busca demora  $O(n)$ , se usarmos vetores não-ordenados também podemos inserir e remover em  $O(1)$ , mas a busca ainda demora  $O(n)$ . Caso tenhamos vetores ordenados buscar demoraria  $O(lgn)$  mas para buscar ou remover levaríamos  $O(n)$ . Veremos aqui uma versão simples de árvores de busca e depois uma versão sofisticada onde todas essas operações levam  $O(lgn)$ .

Uma árvore binária de busca (ABB) é uma árvore binária e, que cada nó contém um elemento de um conjunto ordenável. Cada nó  $r$ , com subárvore esquerda  $TE$  e direita  $TD$  satisfaz a seguinte propriedade:

- 1.esquerda  $< r$  para todo elemento  $e \in TE$ ;
2. direita  $> r$  para todo elemento  $\in TD$ ;



##### a) Implementação:

```

1 typedef struct no *p_no;
2
3 struct no {
4     int chave;
5     p_no esq, dir, pai;
6
7 };
  
```

Definimos a struct no que guarda um inteiro e ponteiros para seus valores, um para o da direita outro para o da esquerda e, se necessário, também podemos guardar um ponteiro para seu valor superior (seu pai).

```

1 p_no buscar(p_no raiz, int chave) {
2     if (raiz == NULL || chave == raiz->chave)
3         return raiz;
4     if (chave < raiz->chave)
5         return buscar(raiz->esq, chave);
6     else
7         return buscar(raiz->dir, chave);
8 }

```

contém a chave, retornando o ponteiro para esse nó (seja o nó encontrado ou NULL). Caso contrário: se a chave procurada for menor que a do nó atual, a busca continua recursivamente pela sub-árvore esquerda; se for maior, a busca prossegue pela sub-árvore direita, otimizando o processo ao descartar metade da árvore a cada passo.

A eficiência de uma busca em árvore binária depende crucialmente do seu balanceamento. Em um cenário ideal, com a árvore bem distribuída, o tempo de busca é logarítmico  $O(\lg n)$ . Contudo, no pior caso, a árvore pode se degenerar em uma estrutura linear, semelhante a uma lista, o que eleva o tempo de busca para linear  $O(n)$ . Apesar disso, em situações comuns onde os elementos são inseridos de forma aleatória, a complexidade média da busca se mantém na eficiente marca de  $O(\lg n)$ .

A função `inserir` adiciona um novo nó a uma árvore binária de busca de forma recursiva. Ela navega pela árvore, movendo-se para a sub-árvore esquerda se a nova chave for menor que a do nó atual, ou para a direita, caso contrário. Ao encontrar uma posição vazia (NULL), ela cria um nó, atribui-lhe o valor da chave e o retorna para ser conectado ao seu nó pai. Este processo garante que o novo elemento seja inserido na posição correta, mantendo a propriedade de ordenação da árvore.

```

1 p_no inserir(p_no raiz, int chave) {
2     p_no novo;
3     if (raiz == NULL) {
4         novo = malloc(sizeof(struct no));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
9     if (chave < raiz->chave)
10        raiz->esq = inserir(raiz->esq, chave);
11    else
12        raiz->dir = inserir(raiz->dir, chave);
13    return raiz;
14 }

```

```

1 p_no minimo(p_no raiz) {
2     if (raiz == NULL || raiz->esq == NULL)
3         return raiz;
4     return minimo(raiz->esq);
5 }

```

que o menor elemento está sempre no extremo esquerdo da árvore. Partindo da raiz, a função

A função `buscar` implementa uma busca recursiva em uma árvore binária de busca para encontrar um nó com uma chave específica. Partindo da raiz, a função primeiro verifica se o nó atual é nulo ou se já

A função `mínimo` encontra o nó com o menor valor em uma árvore binária de busca usando recursão. Ela explora a propriedade de

desce recursivamente através dos filhos da esquerda (esq) até encontrar um nó que não possua mais um filho esquerdo.

A função `sucessor` determina qual é o próximo nó em ordem crescente a partir de um nó  $x$  em uma árvore binária de busca. Ela opera com base em duas regras: primeiramente, se o nó  $x$  tiver uma subárvore direita, seu sucessor será o menor elemento contido nessa subárvore. Caso contrário, se não houver uma subárvore direita, o sucessor será o primeiro ancestral de  $x$  que o contém em sua própria subárvore esquerda, uma tarefa que é delegada à função auxiliar `ancestral_a_direita`:

```
1 p_no sucessor(p_no x) {
2     if (x->dir != NULL)
3         return minimo(x->dir);
4     else
5         return ancestral_a_direita(x);
6 }
```

retornado. Caso contrário, se o nó for um filho direito, a busca continua subindo recursivamente, até que o sucessor seja encontrado ou a raiz da árvore seja alcançada, indicando que não há sucessor.

A função `remover_rec` apaga um nó de uma árvore binária de busca de forma recursiva. Primeiro, ela navega pela árvore para encontrar o nó com a chave desejada. Ao encontrá-lo, a remoção depende do número de filhos: se o nó tem zero ou um filho, ele é simplesmente substituído pelo seu único filho (ou `NULL`). Se o nó tem dois filhos, o método mais comum, indicado pela chamada `remover_sucessor`, é substituir o valor do nó pelo valor do seu sucessor e, em seguida, remover o nó sucessor da árvore, preservando a propriedade de ordenação.

```
1 p_no sucessor(p_no x) {
2     if (x->dir != NULL)
3         return minimo(x->dir);
4     else
5         return ancestral_a_direita(x);
6 }
```

Essa função, opera subindo na árvore a partir do nó  $x$ , utilizando ponteiros para o nó pai. A cada nível, ela verifica se o nó atual é um filho esquerdo. Se for, significa que seu pai é o sucessor procurado e ele é

```
1 p_no remover_rec(p_no raiz, int chave) {
2     if (raiz == NULL)
3         return NULL;
4     if (chave < raiz->chave)
5         raiz->esq = remover_rec(raiz->esq, chave);
6     else if (chave > raiz->chave)
7         raiz->dir = remover_rec(raiz->dir, chave);
8     else if (raiz->esq == NULL)
9         return raiz->dir;
10    else if (raiz->dir == NULL)
11        return raiz->esq;
12    else
13        remover_sucessor(raiz);
14    return raiz;
15 }
```

```

1 void remover_sucessor(p_no raiz) {
2     p_no min = raiz->dir; // será o mínimo da subárvore direita
3     p_no pai = raiz; // será o pai de min
4     while (min->esq != NULL) {
5         pai = min;
6         min = min->esq;
7     }
8     if (pai->esq == min)
9         pai->esq = min->dir;
10    else
11        pai->dir = min->dir;
12    raiz->chave = min->chave;
13 }

```

ponteiros de seu pai, e por fim copia o valor do sucessor para o nó que originalmente se desejava apagar. Desta forma, o valor do nó é efetivamente removido enquanto a estrutura e a propriedade de ordenação da árvore são mantidas.

```

1 void imprime(p_no raiz) {
2     if (raiz != NULL) {
3         imprime(raiz->esq);
4         printf("%d ", raiz->chave);
5         imprime(raiz->dir);
6     }
7 }

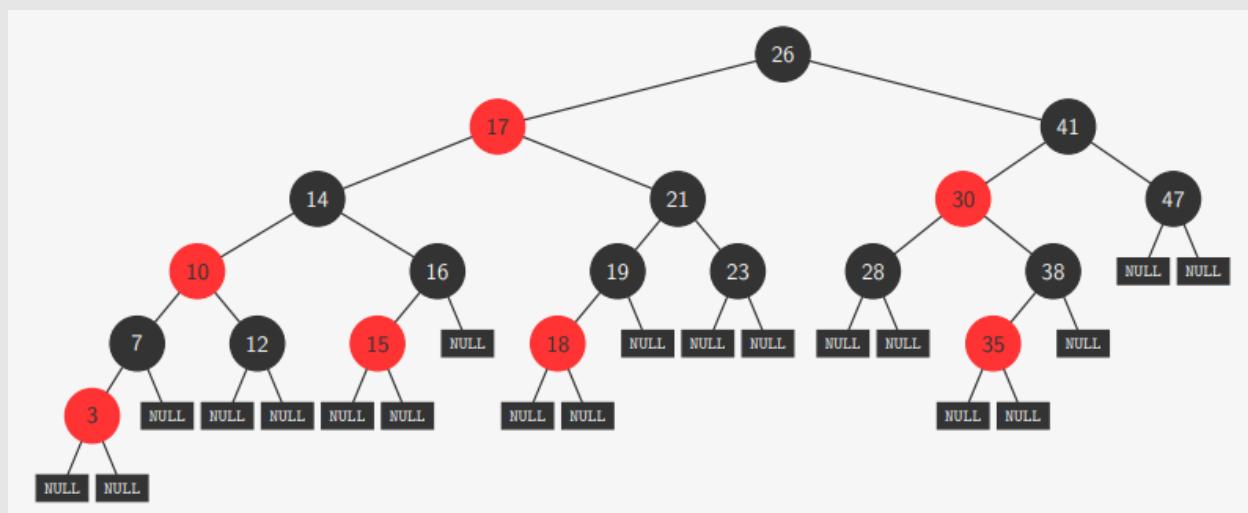
```

de toda a sua sub-árvore direita. Quando aplicada a uma Árvore Binária de Busca, essa sequência garante que os valores sejam impressos em ordem crescente.

A função `remover_sucessor` é uma rotina auxiliar para o caso de remoção de um nó com dois filhos. Ela encontra o nó sucessor (o menor elemento na sub-árvore direita do nó a ser removido), remove este sucessor de sua posição original religando os

A função `imprime` exibe no terminal todos os valores de uma árvore binária utilizando uma abordagem recursiva. Para qualquer nó, ela primeiro executa a impressão de toda a sua sub-árvore esquerda, depois imprime o valor do próprio nó e, por fim, executa a impressão

### **15. Árvores rubro-negras esquerdistas:**



Uma árvore rubro-negra esquerdista é uma ABB tal que:

1. Todo nó é preto ou vermelho;
2. A raiz é preta;
3. As folhas são NULL e tem cor preta;

4. Se um nó é vermelho seus dois filhos são pretos e ele é o filho esquerdo do seu pai;
  5. Em cada nó, todo caminho dele para uma de suas folhas descendentes tem a mesma quantidade de nós pretos (altura negra do nó);
- Seja  $bh$  a altura-negra da árvore, sabe-se que ela tem pelo menos  $2^{bh}-1$  nós não nulos, prova-se por indução. A altura negra é pelo menos metade da altura  $h$  da árvore, não existe nó vermelho com filho vermelho.

a) Alterando a struct e testando a cor:

```

1 enum cor {VERMELHO, PRETO};
2
3 typedef struct no *p_no;
4
5 struct no {
6     int chave;
7     enum cor cor; // campo cor do tipo enum cor
8     p_no esq, dir;
9 };

```

Este código define a estrutura de um nó para uma árvore rubro negra. A estrutura do nó é praticamente a mesma de uma árvore binária de busca normal, contendo um campo para a chave (o valor) e dois ponteiros, esq e dir, para os filhos. A diferença fundamental é a adição de um novo campo, cor, que armazena a cor do nó (VERMELHO ou PRETO). Este campo de cor é a base para

valor) e dois ponteiros, esq e dir, para os filhos. A diferença fundamental é a adição de um novo campo, cor, que armazena a cor do nó (VERMELHO ou PRETO). Este campo de cor é a base para

### Enumerações em C (enum):

A sintaxe geral é a seguinte:

```
enum nome_do_tipo { CONSTANTE1, CONSTANTE2, ... };
```

A lógica é:

1. **enum nome\_do\_tipo:** O programa primeiro declara um novo tipo de dado customizado com a palavra-chave enum. nome\_do\_tipo será o nome desse novo tipo (no exemplo, cor)
2. **{CONSTANTE1, CONSTANTE2, ...}:** Dentro das chaves, você define uma lista de identificadores. Cada um deles se torna uma constante inteira nomeada. Eles são os únicos valores que uma variável do tipo enum nome\_do\_tipo pode ter.
3. **Valores:** O compilador associa um número inteiro a cada constante, começando em 0. No exemplo enum cor {VERMELHO, PRETO}, VERMELHO vira uma constante para o valor 0, e PRETO vira uma constante para o valor 1. Essencialmente, enum cria apelidos legíveis para números inteiros.

os algoritmos de auto-balanceamento das Árvores Rubro-Negras, que garantem que a árvore nunca se degenera, mantendo as operações eficientes.

```
1 int ehVermelho(p_no x) {  
2     if (x == NULL)  
3         return 0;  
4     return x->cor == VERMELHO;  
5 }
```

A função ehVermelho é uma pequena função auxiliar que verifica de forma segura se um nó é da cor vermelha. Ela primeiro trata o caso especial de um nó NULL, que por definição em uma Árvore Rubro-Negra é considerado preto, retornando 0 (falso). Para um nó válido, ela simplesmente retorna o resultado da comparação do campo cor do nó com a constante VERMELHO, resultando em 1 (verdadeiro) se o nó for vermelho e 0 (falso) caso contrário. Para as próximas funções usaremos a função ehPreto, ela é exatamente igual a essa só mudando sua cor.

b) Rotações:

Imagine que temos a seguinte árvore, que já está válida e balanceada:

Situação Inicial (Antes do Problema), árvore válida:

(10)

\

(20)

Agora, vamos inserir o número 30. Lembre-se que todo novo nó é inserido inicialmente com a cor vermelha, o 30 é maior que 10 e maior que 20, então ele entra como filho direito do 20. A árvore fica assim:

(10)

\

(20)

\

(30)

Agora temos um nó pai vermelho (20) com um filho também vermelho (30). Isso viola a regra fundamental da árvore. Além disso, a árvore está virando uma "linha", exatamente o que queremos evitar, é aqui que a rotação se torna necessária para consertar a estrutura, para corrigir o algoritmo de rotação realiza duas operações: Primeiro, aplicamos uma rotação para a esquerda no nó 10. O filho direito (20) sobe para se tornar a nova raiz, e o 10 desce para se tornar seu filho esquerdo.

(20)

/ \

(10) (30)

Para finalizar o balanceamento, o algoritmo troca as cores do "pai" e do "avô" originais, o nó 20 que era vermelho fica preto e o nó 10 que era preto fica vermelho.



```
1 p_no rotaciona_para_esquerda(p_no raiz) {
2     p_no x = raiz->dir;
3     raiz->dir = x->esq;
4     x->esq = raiz;
5     x->cor = raiz->cor;
6     raiz->cor = VERMELHO;
7     return x;
8 }
```

Aplicando a função **rotaciona para esquerda**, o nó raiz que entra na função é o 10 (Preto). A linha 2 define x como o filho direito, o nó 20 (Vermelho). Em seguida, o código réarranja os ponteiros: o 10 passa a apontar

para o filho esquerdo de 20 (que era nulo), e o 20 assume o 10 como seu novo filho esquerdo. É neste momento que a 'virada' da rotação acontece, transformando 20 no novo topo da subárvore. Depois, as cores são ajustadas para resolver a violação da regra: o 20 (x) assume a cor original do 10, tornando-se Preto, e o 10 (raiz) é forçado a se tornar Vermelho. Finalmente, a função retorna o ponteiro para x (o nó 20), que passa a ser a nova raiz balanceada daquela seção da árvore.

A função **rotaciona para direita** é exatamente análoga e espelhada à rotaciona para esquerda.

```
1 p_no rotaciona_para_direita(p_no raiz) {
2     p_no x = raiz->esq;
3     raiz->esq = x->dir;
4     x->dir = raiz;
5     x->cor = raiz->cor;
6     raiz->cor = VERMELHO;
7     return x;
8 }
```

```
1 void sobe_vermelho(p_no raiz) {
2     raiz->cor = VERMELHO;
3     raiz->esq->cor = PRETO;
4     raiz->dir->cor = PRETO;
5 }
```

A função **sobe vermelho**, também conhecida como 'inversão de cores', é uma operação de manutenção usada em Árvores Rubro-Negras. Ela é aplicada a um nó preto (raiz) que possui dois filhos vermelhos. A

operação simplesmente inverte as cores dos três nós: o nó pai se torna vermelho, e seus dois filhos se tornam pretos. Essa manobra é a solução padrão para quando uma inserção causa um conflito "vermelho-vermelho" e o "tio" do novo nó também é vermelho, passando a potencial violação um nível acima na árvore para ser resolvida.

c) Inserção:

O principal problema que pode ocorrer após a inserção de um novo nó (que é sempre vermelho) é a violação da regra de que um nó vermelho não pode ter um filho vermelho. Quando isso acontece, o algoritmo de correção olha para o "tio" do nó recém-inserido (o irmão de seu pai) para decidir o que fazer. Se o tio também for vermelho, a solução é simples: uma inversão de cores, que torna o pai e o tio pretos e o avô vermelho, empurrando a complexidade para o nível de cima da árvore. Se o tio for preto, a solução é mais complexa e exige rotações para reorganizar a estrutura e eliminar o conflito vermelho-vermelho.

Um segundo tipo de problema, específico da implementação mostrada (uma Árvore Rubro-Negra "esquerdista"), é a existência de um "link vermelho pendendo para a direita". Nessa variação da árvore, é proibido que um nó tenha um filho direito vermelho se o esquerdo não for. Qualquer situação que crie essa estrutura "torta" para a direita é considerada um desbalanceamento temporário que deve ser corrigido imediatamente com uma rotação para a esquerda, transformando-o em um link vermelho à esquerda, que é a estrutura permitida.

A função ao lado é a inserção seguida da correção da árvore, executado em cada nó no caminho de volta da inserção. Ele aplica uma sequência de três regras para garantir o balanceamento: primeiro, ele usa uma rotação para a esquerda para consertar qualquer "link vermelho" que esteja incorretamente pendendo para a direita.

```
1 p_no inserir_rec(p_no raiz, int chave) {
2     p_no novo;
3     if (raiz == NULL) {
4         novo = malloc(sizeof(struct no));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         novo->cor = VERMELHO;
8         return novo;
9     }
10    if (chave < raiz->chave)
11        raiz->esq = inserir_rec(raiz->esq, chave);
12    else
13        raiz->dir = inserir_rec(raiz->dir, chave);
14    // corrige a árvore
15    if (ehVermelho(raiz->dir) && ehPreto(raiz->esq))
16        raiz = rotaciona_para_esquerda(raiz);
17    if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq))
18        raiz = rotaciona_para_direita(raiz);
19    if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir))
20        sobe_vermelho(raiz);
21    return raiz;
22 }
```

Em segundo lugar, ele usa uma rotação para a direita para resolver o caso de uma "linha" de dois nós vermelhos seguidos à esquerda. Por último, ele usa a inversão de cores (sobe vermelho) para tratar o caso de um nó que possui dois filhos vermelhos. Essa sequência de passos, aplicada recursivamente, garante que todas as propriedades da Árvore Rubro-Negra sejam restauradas após a inserção.

## **16. Filas de prioridade e Heap:**

Filas de prioridade são estruturas de dados com duas operações básicas, inserir novos elementos e remover o elemento com maior chave(heap). Pilhas são como filas de prioridade, em que o elemento com maior chave é sempre o último a ser inserido, filas também são, mas sua chave é sempre o primeiro a ser inserido.

Várias vezes trocaremos dois elementos de posição, para isso usaremos a seguinte função:

```
1 void troca(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

Ou seja, `troca(&v[i], &v[j])` troca os valores de `v[i]` e `v[j]`.

### a) Implementação:

As duas estruturas, Item e FP, trabalham em conjunto para formar a fila de prioridade. A struct Item define a unidade básica de dados, ou seja, como é um único elemento que será armazenado; ele possui um nome para identificação e uma chave que determina sua prioridade. Já a struct FP define a fila como um todo, agindo como o contêiner que gerencia esses itens. Ela contém o ponteiro `*v` para o vetor onde os Items são efetivamente guardados, e as variáveis `n` e `tamanho` para controlar, respectivamente, a quantidade atual de elementos e a capacidade máxima da estrutura.

```
1 typedef struct {  
2     char nome[20];  
3     int chave;  
4 } Item;  
5  
6 typedef struct {  
7     Item *v; // vetor de Items alocado dinamicamente  
8     int n, tamanho; // n: qtde de elementos, tamanho: qtde alocada  
9 } FP;  
10  
11 typedef FP * p_fp;
```

```
1 p_fp criar_fila prio(int tam) {  
2     p_fp fprio = malloc(sizeof(FP));  
3     fprio->v = malloc(tam * sizeof(Item));  
4     fprio->n = 0;  
5     fprio->tamanho = tam;  
6     return fprio;  
7 }
```

A função `criar_fila prio` executa todo o processo de criação e inicialização de uma nova fila de prioridade. Essencialmente, ela realiza duas alocações de memória dinâmicas: uma para a estrutura principal FP que serve como o contêiner, e outra, maior,

para o vetor v que irá de fato armazenar os elementos (Items), com a capacidade definida pelo parâmetro tam. Após alocar a memória, a função configura os valores iniciais da fila, ajustando o contador de elementos n para 0 (pois ela começa vazia) e o campo tamanho para a capacidade máxima solicitada. Por fim, a função retorna o ponteiro para esta estrutura recém-criada e pronta para uso.

```

1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4 }

```

A função **insere** executa a adição de um novo item à fila de prioridade. Ela o faz inserindo o item diretamente na primeira posição disponível no final do vetor interno

e, em seguida, incrementa o contador de elementos (n) para registrar o aumento no tamanho da fila.

A função **extrai\_maximo** primeiro realiza uma busca linear em todo o vetor para localizar o índice do item com a maior chave de prioridade. Em seguida, ela troca este item de maior prioridade com o último item do vetor. Finalmente, decrementa o contador de elementos da fila para efetivamente "remover" o item máximo, e o retorna ao chamador da função.

```

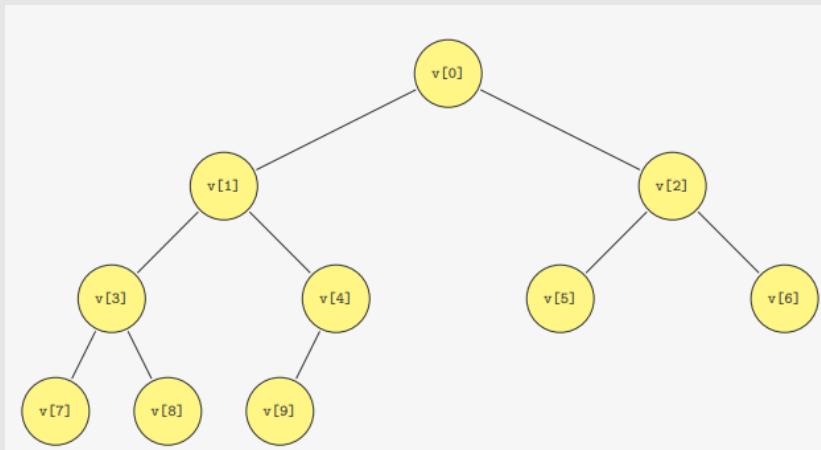
1 Item extrai_maximo(p_fp fprio) {
2     int j, max = 0;
3     for (j = 1; j < fprio->n; j++)
4         if (fprio->v[max].chave < fprio->v[j].chave)
5             max = j;
6     troca(&(fprio->v[max]), &(fprio->v[fprio->n-1]));
7     fprio->n--;
8     return fprio->v[fprio->n];
9 }

```

Esta implementação, embora funcional para um vetor não ordenado, é ineficiente. A busca linear tem uma complexidade de tempo de  $O(n)$ , o que significa que o custo da operação cresce proporcionalmente com o número de elementos na fila. Implementações de filas de prioridade otimizadas, como as baseadas em Heaps, executam esta mesma operação com uma complexidade de tempo de  $O(\log n)$ , sendo drasticamente mais rápidas para grandes volumes de dados.

#### b) Árvores binárias completas:

Uma Árvore Binária Completa é aquela em que todos os níveis, exceto possivelmente o último, estão completamente preenchidos, e todos os nós no último nível estão o mais à esquerda possível. Essa estrutura regular, permite equacionar as posições da árvore e utilizar vetores para representá-la.



Se o nó atual tiver índice  $i$  e assumindo o inicial como sendo 0:  
- $\text{Pai}(i) = (i-1) / 2$ ;  
- $\text{Filho esquerdo} = 2i + 1$ ;  
- $\text{Filho direito} = 2i + 2$ ;

A representação de árvores binárias completas por vetores é altamente vantajosa porque elimina a necessidade de ponteiros, gerando eficiência de memória e excelente localidade de dados, o que acelera o acesso e aproveita o *cache* do processador. Além disso, essa técnica proporciona acesso rápido (1) a qualquer nó (pai, filho esquerdo ou direito) através de simples cálculos de índice, diferentemente da representação por ponteiros, que exige percorrer a estrutura.

### c) Heap de máximo:

Nessa estrutura os filhos são sempre menores ou iguais ao pai, ou seja, a raiz é o máximo. Note que não é uma árvore binária de busca, os dados são bem menos estruturados, pois estamos interessando apenas no valor máximo.

```

1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4     sobe_no_heap(fprio, fprio->n - 1);
5 }
6
7 #define PAI(i) ((i-1)/2)
8
9 void sobe_no_heap(p_fp fprio, int k) {
10    if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
11        troca(&fprio->v[k], &fprio->v[PAI(k)]);
12        sobe_no_heap(fprio, PAI(k));
13    }
14 }
```

O código implementa a inserção em um Max Heap representado por um vetor. A função insere adiciona um novo Item ao final do vetor interno (fprio->v) na posição fprio->n, e imediatamente incrementa o contador de elementos (fprio->n++). Em seguida, ela chama a função auxiliar sobe\_no\_heap para reestabelecer a propriedade do Max Heap: o novo nó inserido é comparado recursivamente com seu nó pai (calculado pela macro PAI(i)), e se o filho tiver uma chave maior que a do pai, eles são trocados (troca) até que a propriedade do Max Heap seja restaurada, garantindo que o maior elemento continue na raiz.

A função implementa a extração do máximo em um Max Heap. A função primeiro armazena o item da raiz (que é o máximo) para retorno (fprio->v[0]). Em seguida, ela move o último elemento do vetor para a raiz, simulando a remoção do máximo, e decremente o tamanho do Heap (fprio->n--). Finalmente, ela chama a função auxiliar desce\_no\_heap a partir da nova raiz (índice 0). Esta função compara recursivamente o nó atual (k) com seu filho de maior chave (maior\_filho), e se o nó atual for menor, realiza uma troca e chama

```

1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
16         if (F_DIR(k) < fprio->n &&
17             fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
18             maior_filho = F_DIR(k);
19         if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
20             troca(&fprio->v[k], &fprio->v[maior_filho]);
21             desce_no_heap(fprio, maior_filho);
22         }
23     }
24 }
```

a si mesma para o filho, restaurando assim a propriedade do Max Heap por todo o caminho da raiz até a base. O valor máximo original é então retornado. A função obedece ao tempo  $O(lgn)$ .

```
1 void muda_prioridade(p_fp fprio, int k, int valor) {
2     if (fprio->v[k].chave < valor) {
3         fprio->v[k].chave = valor;
4         sobe_no_heap(fprio, k);
5     } else {
6         fprio->v[k].chave = valor;
7         desce_no_heap(fprio, k);
8     }
9 }
```

A função muda prioridade permite a atualização da chave de um item no índice para um novo valor, garantindo a manutenção da propriedade do Heap. Após atualizar a chave, a

função verifica se o novo valor é maior que o anterior: se for, chama sobe\_no\_heap para mover o item em direção à raiz; se for menor, chama desce\_no\_heap para movê-lo em direção às folhas. Este mecanismo de checagem condicional e reposicionamento garante que o Max Heap permaneça válido com eficiência  $O(lgn)$ .

## 17. Ordenação:

### a) Bubblesort:

O Bubble Sort é um algoritmo de ordenação simples que percorre repetidamente uma lista, compara elementos adjacentes e os troca de lugar se estiverem na ordem errada. Esse processo é repetido várias vezes, fazendo com que os maiores (ou menores) elementos se alternem/borbulhem gradualmente para o final (ou início) da lista. O algoritmo termina quando uma passagem inteira pela lista é concluída sem que nenhuma troca seja necessária, indicando que a lista está ordenada.

```
1 void bubblesort(int *v, int n) {
2     int i, j;
3     for (i = 0; i < n - 1; i++)
4         for (j = n - 1; j > i; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j-1], &v[j]);
7 }
```

A função possui dois laços. O laço interno (com  $j$ ) move-se do fim do vetor em direção ao índice  $i$ . Ele avalia itens vizinhos: se  $v[j]$  for inferior a  $v[j-1]$  troca e na linha 6 chama a função da aula anterior que inverte suas posições. O laço externo (com  $i$ ) repete esse processo, com um trecho menor, até que tudo

esteja classificado.

No pior caso e, também no caso médio, toda comparação gera uma troca de comparações e trocas em  $O(n^2)$ .

b) Ordenação por inserção:

A ordenação por inserção trata a coleção como duas partes: uma ordenada (no começo) e outra desordenada. O algoritmo seleciona o primeiro item da parte desordenada e o "insere" na sua localização exata dentro da parte ordenada. Para criar espaço, ele desloca para a direita todos os elementos da parte ordenada que são maiores que o item selecionado. Esse procedimento avança, aumentando a parte ordenada, até que nenhum item reste na parte desordenada.

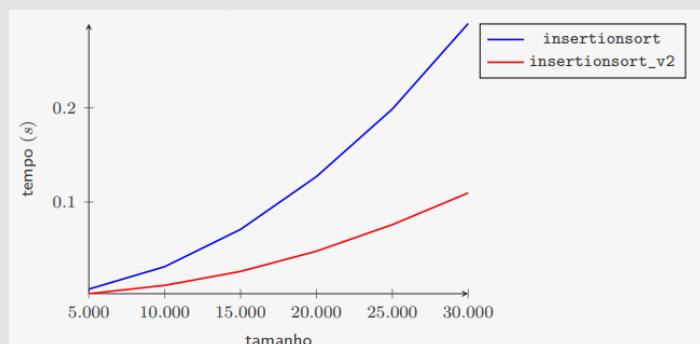
```
1 void insertionsort(int *v, int n) {
2     int i, j;
3     for (i = 1; i < n; i++)
4         for (j = i; j > 0; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```

O laço externo (com  $i$ ) avança a partir do segundo elemento ( $i = 1$ ), tratando a parte do vetor de 0 até  $i-1$  como já classificada. O laço interno (com  $j$ ) pega o elemento  $v[i]$  e o "insere" nessa parte ordenada. Para isso, ele compara  $v[j]$  com seu antecessor  $v[j-1]$  (linha 5) e, se  $v[j]$  for menor, a função troca (linha 6) os inverte de lugar. O laço interno repete esse processo ( $j--$ ), movendo o elemento para a esquerda até que ele chegue ao início do vetor ou encontre um elemento que seja menor ou igual a ele, garantindo que o sub-vetor de 0 a  $i$  fique ordenado.

Podemos otimizar o algoritmo de duas maneiras. A primeira seria para no loop interno com  $j$  assim que seu valor for maior ou igual ao seu antecessor, já que a parte anterior já está ordenada e não precisamos mais verificar nada. A segunda otimização é a mais eficiente e que, de fato, melhora muito o código, ao invés de usar a função troca, armazenamos o valor de  $v[i]$  em uma variável temporária  $t$  e, no loop interno, apenas deslocamos os valores maiores para a direita e, no final, o valor  $t$  é colocado na posição correta.

```
1 void insertionsort_v2(int *v, int n) {
2     int i, j, t;
3     for (i = 1; i < n; i++) {
4         t = v[i];
5         for (j = i; j > 0 && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

No pior caso, ambos os algoritmos para atribuições e comparações usufrui-se de tempo  $O(n^2)$ . No caso médio é a metade disso.



O gráfico ao lado, mostra uma comparação e demonstra o ganho enorme de eficiência com as alterações feitas.

c) Ordenação por seleção:

A Ordenação por Seleção (Selection Sort) funciona construindo uma porção ordenada do vetor, elemento por elemento, a partir do início. O algoritmo executa iterações sucessivas sobre a parte não ordenada. Em cada iteração, ele realiza uma busca linear para identificar o elemento de menor valor presente na sublista não ordenada. Após a identificação, este elemento mínimo é trocado de posição com o primeiro elemento da sublista não ordenada, garantindo assim que ele seja movido para sua posição final correta. O processo se repete, com a fronteira entre as partes ordenada e não ordenada avançando em uma posição a cada ciclo, até que todos os elementos estejam posicionados corretamente e o vetor esteja totalmente ordenado.

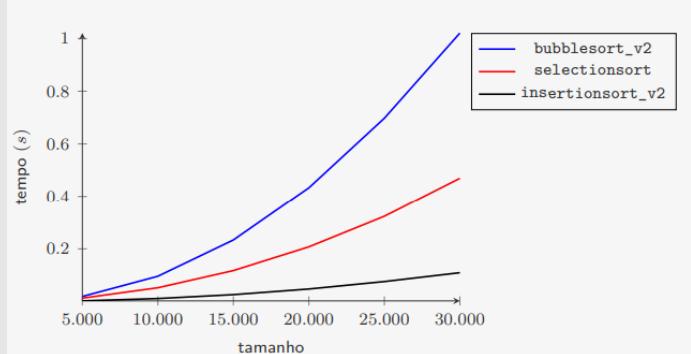
```

1 void selectionsort(int *v, int n) {
2     int i, j, min;
3     for (i = 0; i < n - 1; i++) {
4         min = i;
5         for (j = i+1; j < n; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

Para  $n = 30.000$ : `selectionsort` leva 4,2 o tempo do `insertionsort_v2`, enquanto `bubblesort_v2` leva 9,3 o tempo do `insertionsort_v2`.

Em cada passo do loop externo, o algoritmo garante que o menor elemento da sublista (não ordenada) é encontrado e colocado na posição  $i$ . O processo é repetido até que o penúltimo elemento esteja em seu lugar.

Gráfico comparativo:



d) Ordenação por seleção(-1):

Iremos agora pensar em um algoritmo que primeiro coloca o elemento máximo na última posição do array ( $v[n - 1]$ ), depois coloca o segundo maior elemento na penúltima posição ( $v[n - 2]$ ) e assim por diante, construindo a parte ordenada do vetor do final para o início.

```

1 int extrai_maximo(int *v, int n) {
2     int max = n - 1;
3     for (j = n - 2; j >= 0; j--)
4         if (v[j] > v[max])
5             max = j;
6     return max;
7 }
```

```

1 int selection_invertido_v2(int *v, int n) {
2     int i, j, max;
3     for (i = n - 1; i > 0; i--) {
4         max = extrai_maximo(v, i + 1);
5         troca(&v[i], &v[max]);
6     }
7 }
```

Essa ordenação por seleção gasta tempo  $O(n^2)$ , mas utilizando um heap podemos extrair o máximo em  $O(\lg k)$  e isso vai melhorar muito nosso algoritmo.

e) Heapsort:

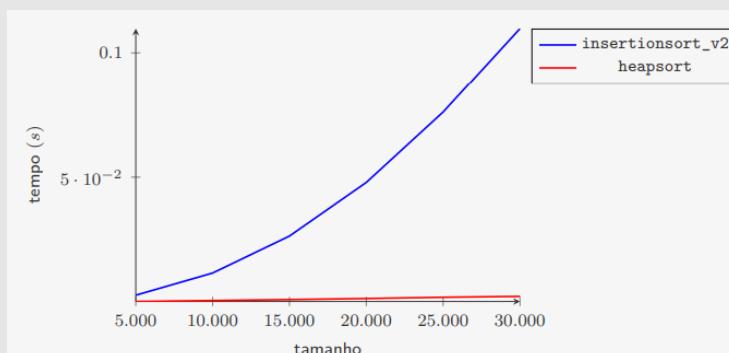
Heapsort funciona construindo a porção ordenada do vetor, elemento por elemento, a partir do final. O algoritmo executa duas fases principais sobre o vetor: primeiro, ele realiza uma etapa inicial para transformar o vetor em uma estrutura de dados de Max Heap(imagem ao lado). Em cada iteração subsequente, ele realiza uma extração do elemento de maior valor, que sempre reside no topo do Heap. Após a identificação, este elemento máximo é trocado de posição com o último elemento da sublista não ordenada, garantindo assim que ele seja movido para sua posição final correta. O processo se repete, com a fronteira entre as partes ordenada e não ordenada recuando em uma posição a cada ciclo e o Heap sendo reestruturado (reorganizado) na sequência, até que todos os elementos estejam posicionados corretamente e o vetor esteja totalmente ordenado.

```

1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, n, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int n) {
16     int k;
17     for (k = n / 2; k >= 0; k--) /* transforma em heap */
18         desce_no_heap(v, n, k);
19     while (n > 1) /* extrai o máximo */
20         troca(&v[0], &v[n - 1]);
21     n--;
22     desce_no_heap(v, n, 0);
23 }
24 }
```

O código implementa o algoritmo de ordenação Heapsort em duas partes. A primeira função (desce no heap) é essencial para restaurar a propriedade de Max Heap, garantindo que o maior elemento desça para sua posição correta na subárvore. A função principal heapsort utiliza esta rotina em duas fases: primeiro, transforma o vetor inteiro em um Max Heap  $O(n)$  ao aplicar desce no heap em todos os nós não folha. Em seguida, na fase de ordenação, repete o processo de extração do máximo, trocando o maior elemento ( $v[0]$ ) com o último elemento não ordenado e, então, reajustando o Heap restante. Este ciclo garante que o Heapsort finalize a ordenação com uma complexidade de tempo eficiente de  $O(n \lg n)$ .

o maior elemento ( $v[0]$ ) com o último elemento não ordenado e, então, reajustando o Heap restante. Este ciclo garante que o Heapsort finalize a ordenação com uma complexidade de tempo eficiente de  $O(n \lg n)$ .



Para  $n = 30000$ , heapsort leva em média  $0.0023s$ , enquanto o insertion sort v2 leva em média  $0.1s$ , 46,3 vezes o tempo do heapsort.

f) Mergesort:

O Merge Sort é um algoritmo de quee segue a estratégia Dividir para Conquistar. Ele funciona dividindo recursivamente o vetor em duas metades menores até que reste apenas um elemento em cada sublista (que já é ordenada porque só tem um elemento). Em seguida, as sublistas ordenadas são gradualmente combinadas (*merged*) para produzir novas sublistas ordenadas maiores, de forma que a união dessas duas partes resulta em uma única lista ordenada. Por dividir o problema de forma equilibrada a cada passo e realizar a intercalação em tempo linear, o Merge Sort garante uma complexidade de tempo de  $O(n \lg n)$ , mas, diferente do Heapsort, ele geralmente requer memória significativa para armazenar temporariamente as sublistas durante o processo de intercalação.

```

1 void merge(int *v, int esq, int meio, int dir) {
2     int aux[MAX]; // podia ter feito alocação dinâmica
3     int i = esq, j = meio + 1, k = 0;
4     // intercala
5     while (i <= meio && j <= dir)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    // copia o resto do subvetor que não terminou
11    while (i <= meio)
12        aux[k++] = v[i++];
13    while (j <= dir)
14        aux[k++] = v[j++];
15    // copia de volta para v
16    for (i = esq, k = 0; i <= dir; i++, k++)
17        v[i] = aux[k];
18 }

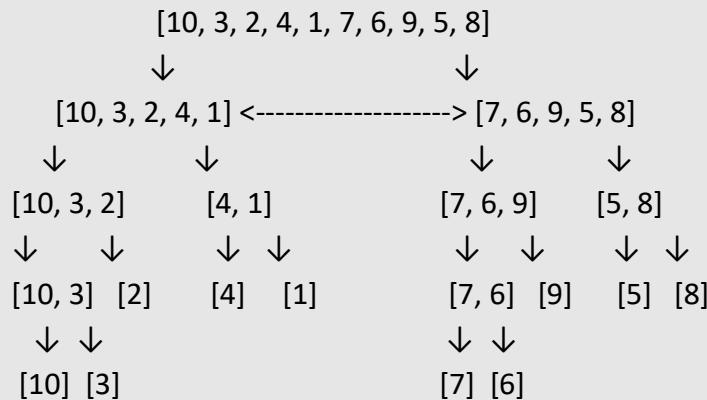
```

```

1 void mergesort(int *v, int esq, int dir) {
2     int meio = (esq + dir) / 2;
3     if (esq < dir) {
4         // divisão
5         mergesort(v, esq, meio);
6         mergesort(v, meio + 1, dir);
7         // conquista
8         merge(v, esq, meio, dir);
9     }
10 }

```

Um exemplo de simulação seria:



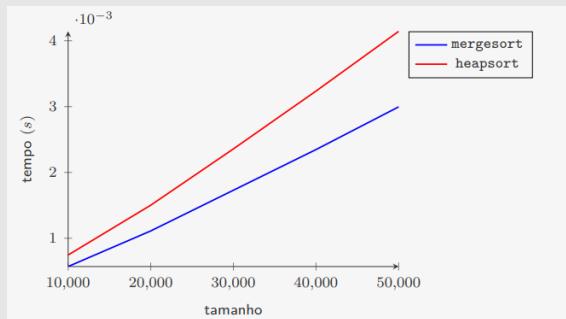
Sublistas	→Lista ordenada	Combinações do Nível 3:	
[10] e [3]	→[3, 10]	[3, 10] e [2]	→[2, 3, 10]
[4] e [1]	→[1, 4]	[1, 4]	→[1, 4]
[7] e [6]	→[6, 7]	[6, 7] e [9]	→[6, 7, 9]
[5] e [8]	→[5, 8]	[5, 8]	→[5, 8]

Combinações do Nível 2:

$$\begin{aligned} [2, 3, 10] \text{ e } [1, 4] &\rightarrow [1, 2, 3, 4, 10] \\ [6, 7, 9] \text{ e } [5, 8] &\rightarrow [5, 6, 7, 8, 9] \end{aligned}$$

Combinação Final (Nível 1):

$$\begin{aligned} [1, 2, 3, 4, 10] \text{ e } [5, 6, 7, 8, 9] &\rightarrow \\ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \end{aligned}$$



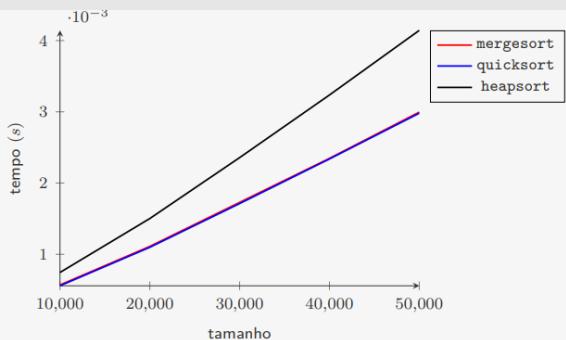
Mergesort também obedece  $O(\lg n)$  e é mais rápido que o heapsort, porém como já falado ele ocupa bem mais memória.

### g) Quicksort:

O Quick Sort é um algoritmo de ordenação muito eficiente que, também, utiliza a estratégia Dividir para Conquistar, mas com foco na partição. O seu funcionamento principal reside em três passos: primeiro, a escolha de um elemento chamado pivô; segundo a partição do vetor em torno desse pivô, movendo todos os elementos menores para um lado e os maiores para o outro, garantindo que o pivô termine em sua posição final correta; e terceiro, a aplicação recursiva do processo de partição nas duas sublistas resultantes (os elementos menores e os elementos maiores).

```
1 void quicksort(int *v, int esq, int dir) {
2     int pos_pivo;
3     if (dir <= esq) return;
4     pos_pivo = partition(v, esq, dir);
5     quicksort(v, esq, pos_pivo - 1);
6     quicksort(v, pos_pivo + 1, dir);
7 }
```

```
1 int partition(int *v, int esq, int dir) {
2     int pivo = v[esq], pos = dir + 1;
3     for (int i = dir; i >= esq; i--) {
4         if (v[i] >= pivo) {
5             pos--;
6             troca(&v[i], &v[pos]);
7         }
8     }
9     return pos;
10 }
```



O quicksort foi levemente mais rápido que o mergesort, mas ainda podemos fazer algumas melhorias nesse código que foi proposto acima, deixando-o ainda mais rápido. Atenta-se ao fato que esse algoritmo é  $O(n^2)$  então como ele pode ser mais eficiente que o HeapSort que obedece a uma lei  $O(n \lg n)$ , se o vetor for uma permutação aleatória de  $n$  elementos, o tempo médio

esperado do QuickSort é  $O(n \lg n)$  já que o pivô vai estar “bem” posicionado, particionando o vetor de uma forma eficiente.

h) Conclusão:

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
HeapSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(1)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	$O(n)$

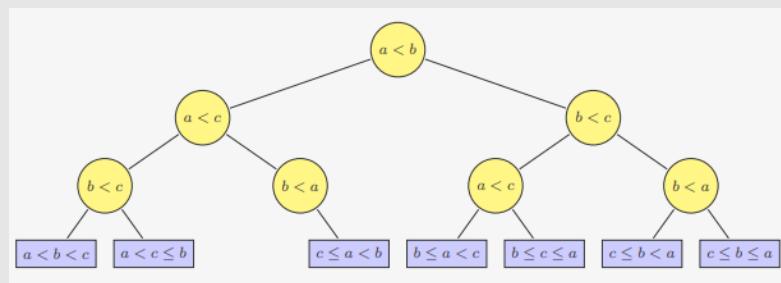
**18. Ordenação em tempo linear:**

Vimos vários algoritmos de ordenação e os melhores tempos estão em  $O(n \lg n)$ , fica a pergunta, dá para fazer melhor que isso?

A resposta é não, se considerarmos algoritmos que usam comparação, que precisam saber apenas se  $v[i] < v[j]$ , algoritmos que são “genéricos”. Porém, a resposta é sim se não usarmos essas comparações, algoritmos que sabem a estrutura da chave (número inteiro com 32 bits).

Um algoritmo de ordenação baseado em comparações recebe uma sequência de  $n$  valores e, precisa decidir qual das  $n!$  permutações é a correta usando apenas comparações entre pares de elementos, pode ordenar int, float, strings, structs, desde que tenha uma função comparação.

Podemos pensar a execução do algoritmo como uma árvore: cada nó representa um teste se  $v[i] < v[j]$ , a subárvore esquerda são as comparações feitas se for verdade e a da direita se for falso, ao lado está a árvore de selections(a,b,c).



Um algoritmo de ordenação é estável se mantém a ordem relativa original dos itens com chaves de ordenação duplicadas, dos que vimos até agora, Insertionsort, BubbleSort e MergeSort o são.

### a) CountingSort:

O Counting Sort é um algoritmo de ordenação que se baseia em contar as ocorrências de cada elemento. Ele é altamente eficiente, com uma complexidade de tempo de  $O(n + k)$ , sendo  $n$  o número de elementos e  $k$  o maior valor no vetor de entrada. Sua principal restrição é que ele só funciona com números inteiros não negativos.

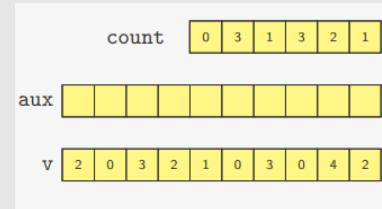
Esse código implementa o

```

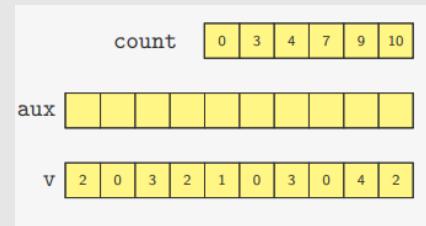
1 #define R 5
2
3 void countingsort(int *v, int l, int r) {
4     int *aux = (int *) malloc((r - l + 1) * sizeof(int));
5     int i, count[R + 1];
6     for (i = 0; i <= R; i++)
7         count[i] = 0;
8     for (i = l; i <= r; i++)
9         count[v[i] + 1]++;
10    for (i = 1; i <= R; i++)
11        count[i] += count[i-1];
12    for (i = l; i <= r; i++) {
13        aux[count[v[i]]] = v[i];
14        count[v[i]]++;
15    }
16    for (i = l; i <= r; i++)
17        v[i] = aux[i-1];
18    free(aux);
19 }
```

Counting Sort visando ordenar um subvetor, definido pelos índices  $l$  (início) e  $r$  (fim), considerando um limite máximo de valor  $R=5$ . A lógica se desenvolve em quatro etapas principais:

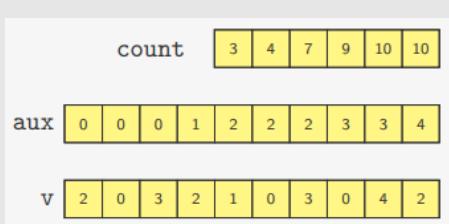
**1: Inicialização e Contagem (Linhas 5-9):** É alocado um vetor auxiliar (`aux`) para o resultado ordenado e um vetor de contagem (`count`) de tamanho  $R+1$  para registrar as ocorrências de cada valor (0 a 5). O código, então, percorre o vetor de entrada `v` no intervalo definido, incrementando a posição em `count` que corresponde ao valor do elemento.



**2: Soma Cumulativa (Linhas 10-11):** O vetor `count` é transformado para armazenar as posições finais dos elementos. Cada `count[i]` passa a guardar o total de elementos que são menores ou iguais a  $i$ . Este passo é crucial, pois é o que permite mapear o valor do elemento para a sua posição exata no vetor de saída.



**3: Distribuição no Vetor Auxiliar (Linhas 12-15):** Esta é a fase de construção do vetor ordenado (`aux`). O algoritmo percorre o *intervalo* do vetor de entrada, `v`, de forma crescente (da esquerda para a direita). Para cada elemento `v[i]`, ele é colocado no índice `count[v[i]]` do vetor auxiliar. Em seguida, o valor em `count[v[i]]` é incrementado.



**4: Cópia (Linhas 16-18):** Por fim, os elementos ordenados do vetor auxiliar (`aux`) são copiados de volta para o vetor original `v`.

### b) RadixSort:

O Radix Sort é um algoritmo de ordenação linear que se destaca por sua capacidade de ordenar números inteiros sem depender de comparações diretas. Sua estratégia é ordenar o conjunto de dados progressivamente, começando pelo dígito menos significativo (unidades) e avançando para os dígitos mais significativos. Para realizar a ordenação em cada dígito, o Radix Sort emprega um algoritmo de ordenação auxiliar estável. A estabilidade é crucial, pois garante que a ordem relativa estabelecida nos dígitos menos significativos seja preservada à medida que os dígitos mais significativos são processados, resultando no vetor completamente ordenado ao final do ciclo.

O termo radix significa a base do sistema numérico, utilizaremos para os exemplos 256 que é 1 byte e o tamanho de um int em C, atenta-se que escolher uma potência de 2 facilita as operações binárias e otimiza o algoritmo. Usar 256 permite que o algoritmo processe o número 32-bit em apenas 4 passadas ( $32/8 = 4$  bytes) ao invés de 32 passadas (se o radix fosse 2, bit a bit). O radix utilizará um algoritmo auxiliar, que é necessariamente estável 4 vezes, ordenando todo mundo com base no 1º byte (menos significativo), posteriormente no 2º e, assim por diante, até o mais significativo.

#### Deslocamento de bits:

- Desloca para a esquerda ( $<<$ ):

Multiplica por  $2^k$ , por exemplo:

$00000101 << 3 == 00101000$  ( $5 << 3 == 40$ )

$01000101 << 3 == 00101000$  ( $69 << 3 == 40$ )

- Desloca para a direita ( $>>$ ):

Divide por  $2^k$ , por exemplo:

$00101000 >> 3 == 00000101$  ( $40 >> 3 == 5$ )

$00101011 >> 3 == 00000101$  ( $43 >> 3 == 5$ )

Para fundamentar nosso algoritmo definiremos:

```
#define bitsword 32
#define bitsbyte 8
#define bytesword 4
#define R 256
#define digit(N,D) (((N) >> (D)*bitsbyte) % R)
```

O último define é fundamental, já que permite ao Radix Sort extrair o D-ésimo "dígito" (byte) de um número.

O código implementa o algoritmo Radix Sort para ordenar um vetor de inteiros (v), que é iterado pelo seu índice de início (l) e fim (r). O coração do algoritmo reside no laço externo, que executa o processo de ordenação uma vez para cada byte do inteiro (quatro vezes no total, definido por bytesword), variando o contador w do byte menos significativo para o mais significativo. A cada iteração, o bloco interno

```
1 void radixsort(int *v, int l, int r) {
2     int *aux = (int *) malloc((r - l + 1) * sizeof(int));
3     int i, w, count[R+1];
4     for (w = 0; w < bytesword; w++) {
5         for (i = 0; i <= R; i++)
6             count[i] = 0;
7         for (i = l; i <= r; i++)
8             count[digit(v[i], w) + 1]++;
9         for (i = 1; i <= R; i++)
10            count[i] += count[i-1];
11         for (i = l; i <= r; i++) {
12             aux[count[digit(v[i], w)]] = v[i];
13             count[digit(v[i], w)]++;
14         }
15         for (i = l; i <= r; i++)
16             v[i] = aux[i-1];
17     }
18     free(aux);
19 }
```

CountingSort no w-ésimo dígito

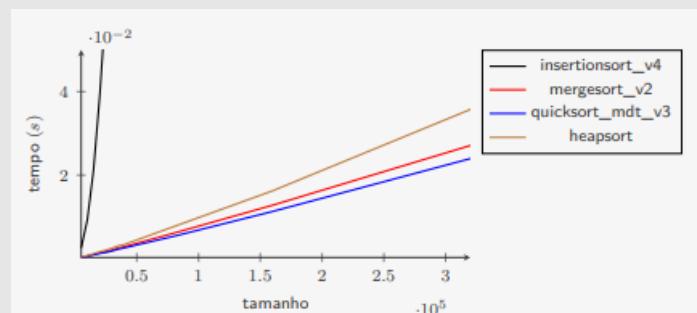
executa uma versão estável do Counting Sort, garantindo que a ordenação estabelecida pelos bytes anteriores seja mantida.

Dentro de cada iteração do laço principal, a primeira etapa do Counting Sort é realizada: o vetor count (dimensionado pelo Radix R=256) é zerado, e em seguida, preenchido com a frequência de cada valor de byte presente no vetor. A chave aqui é a macro digit(v[i], w), que extrai o w-ésimo byte do elemento para ser usado como índice na contagem. Na etapa seguinte, o vetor count é transformado em uma soma cumulativa, de modo que cada count[i] passe a indicar a posição exata onde os elementos com aquele valor de byte (ou menor) devem ser colocados no vetor de saída.

Finalmente, a última fase do Counting Sort, a distribuição, é executada para construir o vetor ordenado. Percorrendo o vetor de entrada, cada elemento é colocado no vetor auxiliar aux na posição indicada pelo vetor count (que funciona como um mapa de índices). Para garantir a estabilidade — crucial no Radix Sort —, o valor em count é incrementado após o uso, preparando o índice para o próximo elemento que possua o mesmo valor de byte. Ao final do Counting Sort, o conteúdo ordenado de aux é copiado de volta para o vetor original v, e o ciclo se repete para o próximo byte. Após a última iteração (o byte mais significativo), o vetor v estará completamente ordenado, e a memória do aux é liberada.

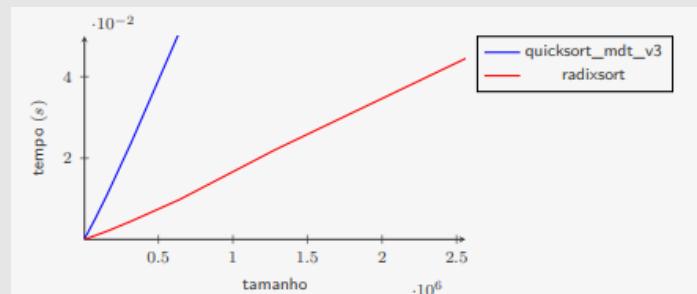
Tempo:  $O(\text{bytesword} \cdot (R + n))$

Se a chave tem  $k$  bits, tempo:  $O\left(\frac{k}{\lg R}(n + R)\right)$



ordenar 5.000 números em 0.034 segundos. O Insertion Sort foi um pouco melhor, ordenando 20.000 números em 0.038 segundos. Já o Quick Sort demonstrou ser muito superior, pois no mesmo tempo máximo (0.05 segundos), ele conseguiu ordenar a impressionante marca de 640.000 números.

O teste de velocidade para ordenação de números, limitado a apenas 0.05 segundos, mostrou uma diferença enorme na capacidade dos algoritmos de lidar com grandes quantidades de dados. O Bubble Sort, por ser o mais lento, só conseguiu



O gráfico ao lado compara o radixsort com o algoritmo mais eficiente acima, o algoritmo agora estudado tem a capacidade de ordenar 256.000.000 números em 0.04s, tornando-se o mais eficiente até agora.

Concluindo:

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	$O(n)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
HeapSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(1)$
RadixSort	$O((n + R) \frac{k}{\lg R})$	$O((n + R) \frac{k}{\lg R})$	$O((n + R) \frac{k}{\lg R})$	$O(R)$

onde  $k$  é o número de bits na chave de ordenação

### **19. Tabela de espalhamento/hashing:**

Uma Tabela de Espalhamento (ou Tabela Hash) é uma estrutura de dados fundamental cujo principal objetivo é permitir o acesso rápido (em tempo constante, ou seja,  $O(1)$  em média) a elementos. Ela opera associando chaves de pesquisa a valores por meio da função de *hashing*, que é o seu mecanismo central: esta função matemática pega um elemento de um conjunto (como uma *string* ou número) e o transforma em um índice dentro de um vetor ou *array* de tamanho conhecido, onde o valor correspondente é armazenado. A eficiência da Tabela Hash reside, portanto, em sua capacidade de calcular diretamente a posição de memória do dado, eliminando a necessidade de longas buscas sequenciais ou complexas. A Tabela de Espalhamento em si é um Tipo Abstrato de Dados (TAD) que gerencia seus elementos acessando-os diretamente através desse vetor, mas precisa lidar com a possibilidade de duas chaves diferentes gerarem o mesmo índice, um fenômeno conhecido como colisão, que requer técnicas específicas (como encadeamento separado ou endereçamento aberto) para ser resolvido.

#### **a) Funções de espalhamento:**

O desempenho ideal de uma Tabela de Espalhamento depende crucialmente da escolha de uma boa função de *hashing*, cujo principal critério é espalhar uniformemente as chaves pela tabela. Idealmente, a probabilidade de qualquer chave gerar um *hash* específico deve ser de aproximadamente  $1/M$  (onde  $M$  é o tamanho da tabela), garantindo que, em média, cada lista (no caso de encadeamento) contenha cerca de  $n/M$  elementos ( $n$  sendo o número total de itens). Existem métodos genéricos, como o Método da Divisão e o Método da Multiplicação, que são amplamente utilizados e funcionam bem na prática. Em um cenário ideal, é possível obter o que se chama de "Hashing Perfeito": se todas as chaves a serem usadas forem conhecidas *a priori*, pode-se encontrar uma função de *hashing* injetora, que mapeia cada chave para um índice único, eliminando completamente as colisões, embora encontrar tais funções possam ser computacionalmente desafiador.

**b) Interpretando chaves:**

Partimos do pressuposto que as chaves precisam ser números inteiros, se não forem reinterpretamos a chave como uma sequência de bits.

Por exemplo, como representaríamos a palavra bala como um número inteiro:

Caractere	Código Binário(8 bits)	Valor Decimal (ASCII)
<b>b</b>	01100010	98
<b>a</b>	01100001	97
<b>l</b>	01101100	108
<b>a</b>	01100001	97

Na base 256:

$$\text{Número} = 98 \cdot 256^3 + 97 \cdot 256^2 + 108 \cdot 256 + 97 \cdot 1 = 1.650.552.465$$

Dessa forma, o número que representaria a palavra bala seria 1.650.552.465. O problema é que temos hashes muito grandes.

**c) Método da divisão:**

O Método da Divisão é uma técnica simples para obter a função de *hashing*, onde o índice  $h(x)$  é o resto da divisão da chave  $x$  pelo tamanho  $M$  da tabela:  $h(x) = x \bmod M$ .

Para que este método funcione bem e espalhe as chaves de forma eficiente, a escolha do  $M$  (tamanho da tabela) é crucial. Não é recomendado escolher  $M$  como uma potência de 2, pois isso faria com que o *hash* dependesse apenas dos bits menos significativos da chave, aumentando a chance de colisões. A prática ideal é escolher  $M$  como um número primo que esteja longe de ser uma potência de 2.

Usando o exemplo da palavra bala, usando  $M = 1783$  (primo longe de potências de 2)

$$H(\text{'bala'}) = 1.650.552.465 \bmod 1783 = 837$$

**d) Método da multiplicação:**

O Método da Multiplicação é uma técnica alternativa para obter a função de *hashing*, onde o índice  $h(x)$  é calculado multiplicando-se a chave  $x$  por um certo valor real  $A$ , obtendo-se a parte fracionária do resultado e, por fim, multiplicando-a pelo tamanho  $M$  da tabela, conforme a fórmula  $h(x) = \lfloor M(A \cdot x \bmod 1) \rfloor$ .

A vantagem desse método é que a posição relativa da chave no vetor não depende do valor de  $M$  (o qual pode ser uma potência de 2, como  $M=1024$ ).

Escolhemos um  $A$  conveniente, geralmente usa-se a razão áurea, que responde bem  $A = (\sqrt{5} - 1)/2$ , sugestão de Knuth.

Usando o exemplo da palavra bala, com  $M = 1024$ :

$$H(\text{'bala'}) = \lfloor 1024 \cdot ((\sqrt{5} - 1)/2 \cdot 1.650.552.465 \bmod 1) \rfloor = 918.$$

### e) Interface do TAD:

```
1 #define MAX 1783
2
3 typedef struct no *p_no;
4
5 struct no {
6     char chave[10];
7     int dado;
8     p_no prox;
9 };
10
11 typedef struct hash *p_hash;
12
13 struct hash {
14     p_no vetor[MAX];
15 };
```

Com base nos trechos de código fornecidos, esta é a implementação de uma Tabela de Espalhamento (*Hash*) com Estrutura de Colisão por Encadeamento Separado. A estrutura de dados é definida com um tamanho fixo de 1783 (MAX), que é um número primo escolhido para otimizar a distribuição das chaves. A tabela principal (struct hash) é um vetor de ponteiros (p\_no vetor[MAX]), onde cada posição aponta para o primeiro elemento de uma lista ligada. Cada elemento, ou Nó (struct no), armazena a chave (string), o dado (informação associada) e um ponteiro prox para o próximo nó da lista.

O mecanismo central da estrutura é a função hash(char \*chave), que converte a chave em uma string no índice numérico do vetor. Essa função utiliza o Método da Divisão, onde o valor de hash é continuamente calculado pela expressão:

$(256 * n + \text{chave}[i]) \% \text{MAX}$   
dentro de um loop.

```
1 int hash(char *chave) {
2     int n = 0;
3     for (int i = 0; i < strlen(chave); i++)
4         n = (256 * n + chave[i]) % MAX;
5     return n;
6 }
7
8 void inserir(p_hash tab, char *chave, int dado) {
9     int n = hash(chave);
10    tab->vetor[n] = inserir_lista(tab->vetor[n], chave, dado);
11 }
12
13 void remover(p_hash tab, char *chave) {
14     int n = hash(chave);
15     tab->vetor[n] = remover_lista(tab->vetor[n], chave);
16 }
```

Esta abordagem é crucial para evitar o overflow de inteiros ao processar strings longas. As operações de manipulação de dados, como inserir e remover, dependem primeiramente do cálculo desse índice. Após obter o índice n, as funções delegam a manipulação dos elementos (inserção ou remoção) para as respectivas funções de lista ligada (inserir\_lista e remover\_lista) que operam no endereço tab->vetor[n], garantindo que o custo médio dessas operações na Tabela Hash seja aproximadamente O(1).

**Observação:** Sabendo a função de hashing o usuário pode facilmente prejudicar o programa ao inserir vários elementos com o mesmo hash, para nos protegermos de um usuário malicioso podemos escolher o hashing aleatoriamente:

1. Fixe p um primo maior que M;

2. Escolha  $a \in \{1, \dots, p\}$  e  $b \in \{0, \dots, p\}$  uniform. Ao acaso;
3. Defina  $h(k) = ((ak+b) \bmod p) \bmod M$ ;
4. Sabemos que essa função espalha bem, é um hashing universal.

**f) Endereçamento aberto:**

Existe uma alternativa para a implementação de tabela de espalhamento, no endereçamento aberto os dados são guardados no próprio vetor e em caso de colisão o elemento é alocado em posições livres da tabela.

Evita percorrer usando ponteiros com alocação e desalocação de memória, se a tabela encher, devemos recriar uma tabela maior.

Para inserir devemos procurar a posição do hashing, se houver espaço guardamos, caso não houver espaço, procuramos a próxima posição livre (Módulo M).

Para fazer a busca é só simular a inserção, calcular a função de hashing e procurar a tabela em sequência procurando pela chave, se encontrar a chave eu devolvo o item correspondente e se encontrar espaço vazio, devolvo NULL.

A remoção é um pouco mais complicada, não podemos apenas remover os elementos da tabela porque isso quebraria a busca. Isso acontece no endereçamento aberto, pois um espaço vazio introduzido no meio de uma sequência de colisões (um "bloco") faria com que a busca parasse, ignorando itens que foram inseridos após aquele ponto, mas que originalmente tinham a mesma posição *hash*.

Para contornar isso, existem três opções principais:

**Opção 1** é realizar o re-hash dos elementos seguintes do bloco, onde removemos os itens até a próxima posição realmente vazia, recalculamos o *hash* de cada um e os reinserimos na tabela. Essa solução pode ser custosa e complexa de implementar.

A **Opção 2** é substituir o item removido por um valor *dummy* (fictício), um marcador especial que indica que o item foi removido, mas que não deve ser confundido com um espaço vazio, permitindo que a busca continue.

**Opção 3** é marcar o item como removido por meio de um campo adicional na estrutura do nó, alcançando o mesmo objetivo da Opção 2 de permitir que a sondagem da busca prossiga. Ao fazer isso as lógicas de inserção e busca devem ser ajustadas para acomodar esse novo estado.

Para a inserção, após calcular o *hash* inicial, o novo item deve ser colocado na primeira posição disponível a partir de que pode ser tanto uma posição completamente vazia quanto uma posição marcada como removida.

Já a busca deve ser modificada para percorrer a tabela a partir do índice, em sequência: ao encontrar um item, deve-se verificar se ele está validamente presente (ou seja, se não foi marcado como removido); é fundamental que a busca passe por cima de qualquer item marcado como removido, parando apenas quando uma posição realmente vazia for encontrada, o que

sinaliza o fim do bloco de sondagem, além de tomar o cuidado necessário para evitar um *loop* infinito.

**g) Hashing duplo:**

É como a sondagem linear, contudo ao invés de pular de 1 em 1 ao encontrar um conflito, chamaremos uma nova função de hash para determinar a sua posição, isso é:

$$h(k,i) = (\text{hash1}(k) + i \cdot \text{hash2}(k)) \bmod M$$

hash2(k) nunca pode ser 0 e deve ser co-primo com M, também ajuda escolher M como uma potência de 2 e fazer que hash2 seja ímpar ou escolher M como um número primo e fazer que  $\text{hash2}(k) < M$ ;

**Sondagem linear - número de acessos médio por busca**

$n/M$	1/2	2/3	3/4	9/10
com sucesso	1.5	2.0	3.0	5.5
sem sucesso	2.5	5.0	8.5	55.5

**Hashing duplo - número de acessos médio por busca**

$n/M$	1/2	2/3	3/4	9/10
com sucesso	1.4	1.6	1.8	2.6
sem sucesso	1.5	2.0	3.0	5.5

**h) Conclusão:**

Hashing é uma boa estrutura de dados para inserir, remover e buscar dados pela sua chave rapidamente, com uma boa função de hashing essas operações levam  $O(1)$ , mas não é uma boa caso quisermos fazer operações relacionadas a ordem das chaves.

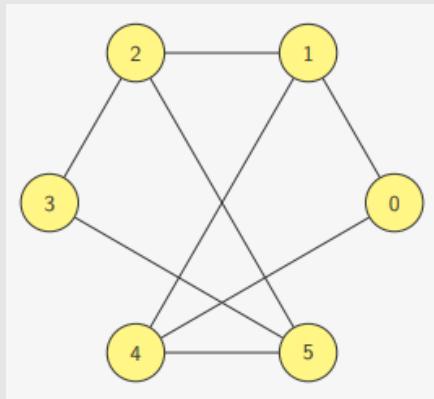
Para a implementação sondagem linear é o mais rápido se a tabela for esparsa, hashing duplo usa melhor a memória, mas gasta mais tempo para computar a segunda função de hash. Encadeamento separado é mais fácil de implementar.

Além disso funções de hashing têm várias outras aplicações, como por exemplo para evitar erros de transmissão, podemos, além de informar uma chave, transmitir o resultado da função de hashing ( dígitos verificadores e sequências de verificação para arquivos como MD5 e SHA) e também usamos hash para segurança, como em um banco ele guarda sua senha como um hash ao invés dela em si, isso evita vazamento de informação em caso de ataque, mas temos que garantir que a probabilidade de duas senhas terem o mesmo hash seja ínfima.

## **21: Grafos(representação):**

Um grafo simplificadamente é um conjunto de objetos ligados entre si, chamamos esses objetos de vértices e, suas conexões de arestas. Visualmente seus vértices são pontos e as arestas são curvas ligando dois deles.

Matematicamente, um grafo  $G$  é um par ordenado  $(V, E)$ , em que  $V$  é o conjunto de vértices do grafo e  $E$  é o conjunto de arestas do grafo, representa-se uma aresta ligando  $u, v \in V$   $\{u, v\}$ , para toda aresta em  $E$  temos que  $u$  é diferente de  $v$  e, existe no máximo, uma aresta  $\{u, v\}$  em  $E$ . Exemplo:



$$V = \{0, 1, 2, 3, 4, 5\};$$
$$E = \{\{0, 1\}, \{0, 4\}, \{5, 3\}, \{1, 2\}, \{2, 5\}, \{4, 5\}, \{3, 2\}, \{1, 4\}\};$$

Dizemos que o vértice 0 é vizinho do vértice 4, dizemos que 0 e 4 são adjacentes, os vértices 0, 1 e 5 formam a vizinhança do vértice 4.

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	1
3	0	0	1	0	0	1
4	1	1	0	0	0	1
5	0	0	1	1	1	0

Vamos representar um grafo por uma matriz de adjacências, se o grafo tem  $n$  vértices eles serão enumerados de 0 até  $n - 1$  e a ordem da matriz é  $n$ , para o exemplo acima sua matriz de adjacência seria:

### a) TAD grafo:

```
1 typedef grafo *p_grafo;
2
3 struct grafo {
4     int **adj;
5     int n;
6 };
```

Definimos a struct `grafo` como uma matriz de adjacência(`**adj`) e seu tamanho(`n`);

```
1 p_grafo criar_grafo(int n) {
2     int i, j;
3     p_grafo g = malloc(sizeof(struct grafo));
4     g->n = n;
5     g->adj = malloc(n * sizeof(int *));
6     for (i = 0; i < n; i++)
7         g->adj[i] = malloc(n * sizeof(int));
8     for (i = 0; i < n; i++)
9         for (j = 0; j < n; j++)
10            g->adj[i][j] = 0;
11    return g;
12 }
```

A função `criar_grafo` está criando o grafo dinamicamente, todos os elementos da matriz são iniciados com o valor 0.

```

1 void destroi_grafo(p_grafo g) {
2     int i;
3     for (i = 0; i < g->n; i++)
4         free(g->adj[i]);
5     free(g->adj);
6     free(g);
7 }

```

A função libera a memória usada no grafo.

Nota-se que os algoritmos para inserir ou remover arestas são triviais, basta mudar o elemento  $u,v$  da matriz de 0 para 1 ou vice-versa.

### b) Problema da rede social:

Podemos entender um grafo como sendo uma rede social, seus vértices são os usuários e a “amizade” entre dois deles são as arestas. Define-se grau de um vértice como sendo seu número de vizinhos.

```

1 int grau(p_grafo g, int u) {
2     int v, grau = 0;
3     for (v = 0; v < g->n; v++)
4         if (g->adj[u][v])
5             grau++;
6     return grau;
7 }

```

A função **mais popular** percorre todo o grafo e retorna o índice daquele vértice que possui o maior grau.

A função **grau**, conta justamente o grau do vértice.

```

1 int mais_popular(p_grafo g) {
2     int max, grau_max, grau_atual;
3     max = 0;
4     grau_max = grau(g, 0);
5     for (int u = 1; u < g->n; u++) {
6         grau_atual = grau(g, u);
7         if (grau_atual > grau_max) {
8             grau_max = grau_atual;
9             max = u;
10        }
11    }
12    return max;
13 }

```

Um problema, também bastante comum é indicar amigos, isso nas redes sociais é normalmente feito como “amigos de amigos”, para indicar novas conexões para o vértice 2 que é conectado com o 5, olharemos as conexões do 5 e recomendaremos para o 2.

### c) Grafos dirigidos (digrafos):

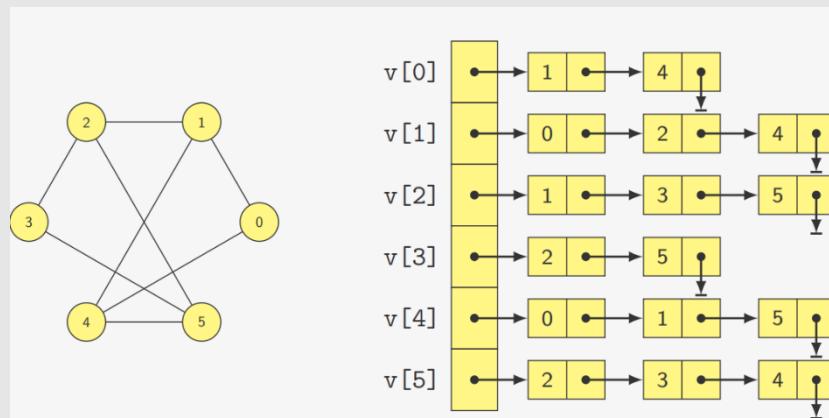
Até então, analisamos os casos em que o vínculo era feito de maneira mútua, agora passaremos a analisar o caso de “seguidores”, esses o vértice 1 pode apontar para o 2, mas o 2 pode não apontar para o 1.

Aí surge a ideia dos grafos dirigidos, eles são conectados através de um conjunto de arcos. Matematicamente um dígrafo  $G$  é um par  $(V, A)$  em que  $V$  é o conjunto de vértices e  $A$  é o conjunto de arcos, representamos um arco ligando  $u, v$  como  $(u, v)$  em que  $u$  é a origem e  $v$  o destino. Podemos ter laços (arcos que partem e chegam no mesmo vértice), sendo que xiste no máximo um arco desse tipo em  $A$ .

Podemos ver um grafo como um dígrafo, já estávamos fazendo isso com a matriz de adjacência. Nota-se que  $\text{adjacencia}[u][v]$  pode ser diferente de  $\text{adjacencia}[v][u]$ .

### d) Grafos com listas de adjacência:

Representando um grafo por listas de adjacência, temos uma lista ligada para cada vértice, a qual armazena quais são os vizinhos do vértice.



Primeiro definimos a struct do nó, ela armazena o valor do vértice( $v$ ) e um ponteiro para o próximo vizinho. Posteriormente a struct do grafo que guarda um array de ponteiros para os nós.

```
1 typedef struct no *p_no;
2
3 struct no {
4     int v;
5     p_no prox;
6 };
7
8 typedef struct grafo *p_grafo;
9
10 struct grafo {
11     p_no *adjacencia;
12     int n;
13 };
```

```

1 p_grafo criar_grafo(int n) {
2     int i;
3     p_grafo g = malloc(sizeof(struct grafo));
4     g->n = n;
5     g->adjacencia = malloc(n * sizeof(p_no));
6     for (i = 0; i < n; i++)
7         g->adjacencia[i] = NULL;
8     return g;
9 }

```

A função criar grafo, apontando todos os ponteiros das listas de adjacência para NULL.

As funções são responsáveis por liberar a memória ocupada pelas listas e grafos.

```

1 void libera_lista(p_no lista) {
2     if (lista != NULL) {
3         libera_lista(lista->prox);
4         free(lista);
5     }
6 }

```

```

1 void destroi_grafo(p_grafo g) {
2     int i;
3     for (i = 0; i < g->n; i++)
4         libera_lista(g->adjacencia[i]);
5     free(g->adjacencia);
6     free(g);
7 }

```

A função auxiliar insere na lista insere um novo nó no início de uma lista ligada, a função insere aresta efetivamente adiciona uma aresta no grafo, sendo ela não direcionada, tendo dois arcos, um em cada direção.

```

1 p_no insere_na_lista(p_no lista, int v) {
2     p_no novo = malloc(sizeof(struct no));
3     novo->v = v;
4     novo->prox = lista;
5     return novo;
6 }

```

```

1 void insere_aresta(p_grafo g, int u, int v) {
2     g->adjacencia[v] = insere_na_lista(g->adjacencia[v], u);
3     g->adjacencia[u] = insere_na_lista(g->adjacencia[u], v);
4 }

```

```

1 p_no remove_da_lista(p_no lista, int v) {
2     p_no proximo;
3     if (lista == NULL)
4         return NULL;
5     else if (lista->v == v) {
6         proximo = lista->prox;
7         free(lista);
8         return proximo;
9     } else {
10        lista->prox = remove_da_lista(lista->prox, v);
11        return lista;
12    }
13 }

```

Esse código faz exatamente o oposto do anterior, a primeira função é uma recursão que remove o primeiro nó com valor v em uma lista ligada, ela retorna um ponteiro para a cabeça da nova lista. A função remove aresta acaba a operação, fazendo-a em ambas as listas de adjacência, removendo a ligação u->v e v->u.

```

1 void remove_aresta(p_grafo g, int u, int v) {
2     g->adjacencia[u] = remove_da_lista(g->adjacencia[u], v);
3     g->adjacencia[v] = remove_da_lista(g->adjacencia[v], u);
4 }

```

### e) Multigrafos:

Essas estruturas permitem arestas paralelas, ou múltiplas, ao invés de um conjunto de arestas temos um “multiconjunto” delas. Pode ser representada por listas de adjacência.

## 20. Grafos(percurso):

Um caminho de  $s$  para  $t$  em um grafo é uma sequência sem repetição de vértices vizinhos começando em  $s$  e terminando em  $t$ . Formalmente um caminho é uma sequência de vértices  $(v_0, v_1, v_2, \dots, v_k)$  em que  $v_0 = s$  e  $v_k = t$ ,  $\{v_i, v_{i+1}\}$  é uma aresta para todo  $0 \leq i \leq k - 1$  e  $v_i$  é diferente de  $v_j$  para todo  $0 \leq i < j \leq k$ .  $k$  é o comprimento do caminho,  $k = 0$  se, e somente se,  $s = t$ .

Um grafo pode ter várias “partes”, chamamos elas de componentes conexas, um par de vértices está na mesma componente conexa se, e somente se, existe caminho entre eles, não há caminho entre vértices de componentes distintas. Um grafo conexo tem apenas uma componente conexa.

### a) Funções:

```
1 int existe_caminho(p_grafo g, int s, int t) {
2     int encontrou, *visitado = malloc(g->n * sizeof(int));
3     for (int i = 0; i < g->n; i++)
4         visitado[i] = 0;
5     encontrou = busca_rec(g, visitado, s, t);
6     free(visitado);
7     return encontrou;
8 }
```

A função `existe_caminho` atua para descobrir se existe um caminho entre um vértice de origem  $s$  e um de destino  $t$ . Para isso, primeiro ela aloca e inicializa m array “visitado” do

tamanho do grafo marcando todos os vértices como 0. Em seguida, ela chama a função `busca_rec`, que realiza a busca recursiva. Após a busca terminar e armazenar o resultado 1 se `encontrou` e 0 se não na variável `encontrou`, a função libera a memória do array `visitado` e retorna o resultado.

Esta é a função `busca_rec`. A função primeiro verifica o caso base: se o vértice atual  $v$  é o alvo  $t$  (linha 3), ela retorna 1 (`encontrou`). Caso contrário, ela marca  $v$  como visitado (linha 5) e, em seguida, itera por todos os possíveis vizinhos  $w$  (linha 6). Se existir uma aresta de  $v$  para  $w$  ( $g->adj[v][w]$ ) e  $w$  ainda não foi visitado (`!visitado[w]`), a função se chama recursivamente para  $w$  (linha 8). Se essa chamada

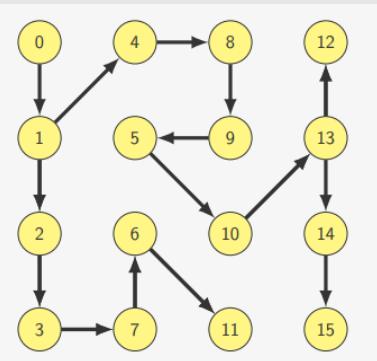
```
1 int busca_rec(p_grafo g, int *visitado, int v, int t) {
2     int w;
3     if (v == t)
4         return 1; // sempre existe caminho de t para t
5     visitado[v] = 1;
6     for (w = 0; w < g->n; w++)
7         if (g->adj[v][w] && !visitado[w])
8             if (busca_rec(g, visitado, w, t))
9                 return 1;
10    return 0;
11 }
```

recursiva encontrar o alvo, ela retorna 1; se o loop terminar sem encontrar um caminho a partir de v (explorou todos os vizinhos e nenhum levou a t), ela retorna 0.

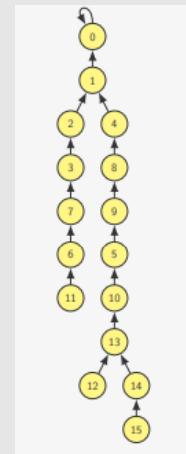
### b) Ciclos em grafos:

Um ciclo em um grafo é uma sequência de vértices sozinhos sem repetição exceto pelo primeiro e o último vértice que são idênticos. Uma floresta é um grafo conexo acíclico, suas componentes conexas são árvores.

Um subgrafo é um grafo obtido a partir da remoção de vértices e arestas, podemos considerar também árvores/florestas que são subgrafos de um grafo dado.



Podemos olhar os caminhos de um vértice s para outros vértices da componente como uma árvore, na verdade as arestas usadas são uma árvore, ela dá um caminho de qualquer vértice até a raiz, basta ir subindo a árvore.



### c) Busca em profundidade com recursão:

A Busca em Profundidade (DFS-Depth-First Search) é um algoritmo usado para explorar todos os nós de um grafo ou de uma árvore. A ideia central do DFS é explorar o mais fundo possível por um caminho antes de voltar atrás (fazer *backtracking*) e tentar um caminho diferente.

```
1 int * encontra_caminhos(p_grafo g, int s) {
2     int i, *pai = malloc(g->n * sizeof(int));
3     for (i = 0; i < g->n; i++)
4         pai[i] = -1;
5     busca_em_profundidade(g, pai, s, s);
6     return pai;
7 }
```

Esse código executa uma Busca em Profundidade (DFS) a partir de um nó inicial s e usa um array pai para armazenar o resultado. O encontra\_caminhos apenas prepara o terreno,

criando o array pai e marcando todos os vértices como "não visitados" (com -1). Em seguida, ele chama a função recursiva busca\_em\_profundidade. Essa função "visita" um nó v marcando quem é seu pai p (na linha `pai[v] = p`). Depois, ela olha para todos os vizinhos de v. Se um vizinho ainda não foi visitado (`pai[vizinho] == -1`), ela "mergulha" recursivamente para visitar aquele vizinho, e o v atual se torna o pai daquele vizinho. O resultado final é que o array pai descreve exatamente a árvore de busca em profundidade, como a desenhada na imagem à direita.

Naquela imagem, a seta de um nó sempre aponta para seu "pai" (ex: a seta do nó 7 aponta para o 3, significando  $pai[7] = 3$ ).

```

1 void busca_em_profundidade(p_grafo g, int *pai, int p, int v) {
2     p_no t;
3     pai[v] = p;
4     for(t = g->adj[v]; t != NULL; t = t->prox)
5         if (pai[t->v] == -1)
6             busca_em_profundidade(g, pai, v, t->v);
7 }
```

#### d) Busca em profundidade usando uma pilha:

Podemos fazer a busca em profundidade usando uma pilha, a cada passo, desempilhamos um vértice não visitado e inserimos os seus vizinhos não visitados a pilha. A versão recursiva do DFS usa a "pilha de chamadas" do próprio sistema para "lembrar" para onde deve voltar. A versão iterativa simplesmente substitui essa pilha de chamadas implícita por uma pilha de dados que nós mesmos controlamos.

```

1 int * busca_em_profundidade(p_grafo g, int s) {
2     int w, v;
3     int *pai = malloc(g->n * sizeof(int));
4     int *visitado = malloc(g->n * sizeof(int));
5     p_pilha p = criar_pilha();
6     for (v = 0; v < g->n; v++) {
7         pai[v] = -1;
8         visitado[v] = 0;
9     }
10    empilar(p, s);
11    pai[s] = s;
12    while(!pilha_vazia(p)) {
13        v = desempilar(p);
14        if (visitado[v]) continue;
15        visitado[v] = 1;
16        for (w = 0; w < g->n; w++)
17            if (g->adj[v][w] && !visitado[w]) {
18                pai[w] = v;
19                empilar(p, w);
20            }
21    }
22    destroi_pilha(p);
23    free(visitado);
24    return pai;
25 }
```

Este código realiza uma DFS usando uma pilha para controlar a ordem de visita e um array visitado para evitar trabalho duplicado. O algoritmo começa empilhando o nó inicial  $s$ . Em seguida, entra em um loop while que, a cada passo, desempilha um vértice  $v$ , o marca como visitado e, em seguida, verifica sua linha inteira na matriz de adjacência. Se encontrar um vizinho  $w$  que ainda não foi visitado ( $g->adj[v][w] \&\& !visitado[w]$ ), ele define  $v$  como o pai de  $w$  e empilha  $w$ . Como a pilha é LIFO, o algoritmo naturalmente "mergulha" no vizinho mais recentemente encontrado, criando o comportamento de profundidade.

#### e) Busca (em largura) usando fila:

A Busca em Largura (BFS) é um algoritmo para percorrer grafos que explora os nós "camada por camada", começando por um nó raiz. Ele primeiro visita todos os vizinhos imediatos do nó (a "Camada 1"), depois todos os vizinhos desses vizinhos (a "Camada 2"), e assim sucessivamente. A Fila é a estrutura de dados perfeita para isso por causa de sua natureza: ao enfileirar os vizinhos de um nó, a fila garante que todos os nós da camada atual (que entraram primeiro) serão visitados antes que o algoritmo comece a visitar os nós da próxima camada (que entraram por

último). Isso força a exploração a se expandir uniformemente "em largura", garantindo que o primeiro caminho encontrado para qualquer nó seja o caminho mais curto.

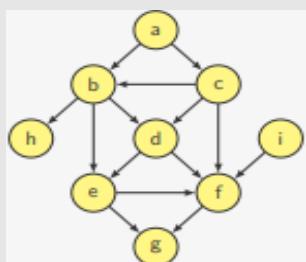
```

1 int * busca_em_largura(p_grafo g, int s) {
2     int w, v;
3     int *pai = malloc(g->n * sizeof(int));
4     int *visitado = malloc(g->n * sizeof(int));
5     p_fila f = criar_fila();
6     for (v = 0; v < g->n; v++) {
7         pai[v] = -1;
8         visitado[v] = 0;
9     }
10    enfileira(f, s);
11    pai[s] = s;
12    visitado[s] = 1;
13    while(!fila_vazia(f)) {
14        v = desenfileira(f);
15        for (w = 0; w < g->n; w++)
16            if (g->adj[v][w] && !visitado[w]) {
17                visitado[w] = 1; // evita repetição na fila
18                pai[w] = v;
19                enfileira(f, w);
20            }
21    }
22    destroi_fila(f);
23    free(visitado);
24    return pai;
25 }
```

qualquer nó a uma distância  $k+1$ , garantindo que o array pai resultante contenha os caminhos mais curtos desde  $s$ .

## 21. Grafos(algoritmos):

Grafos são usados em arquivos Makefiles, para realizar uma tarefa, precisamos realizar várias antes das quais a primeira é dependente, vamos modelar usando um dígrafo:

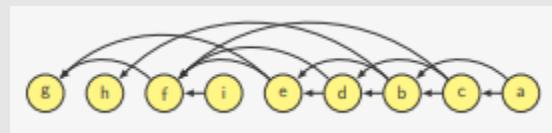


Na imagem ao lado eu estou basicamente dizendo que para realizar a tarefa  $a$ , antes eu preciso realizar  $b$  e  $c$ ...

Um dígrafo acíclico (DAG) é um dígrafo que não contém ciclos dirigidos. As tarefas podem ser realizadas se, e somente se, o dígrafo de dependências é um DAG.

Em qual ordem devemos organizar essas tarefas

- 1)  $g$  e  $h$  não dependem de outra tarefa;
- 2)  $f$  depende apenas de  $g$ ;
- 3)  $i$  depende apenas de  $f$ ;
- 4)  $e$  depende apenas de  $f$  e  $g$ ;
- 5)  $d$  depende apenas de  $e$  e  $f$ ;
- 6)  $b$  depende apenas de  $h$ ,  $e$  e  $d$ ;
- 7)  $c$  depende apenas de  $b$ ,  $d$  e  $f$ ;
- 8)  $a$  depende apenas de  $b$  e  $c$ ;



Este código implementa BFS usando uma Fila para garantir uma exploração em "camadas". Ele inicia os arrays pai e visitado, enfileira o nó inicial  $s$  e o marca como visitado. O loop while principal retira um vértice  $v$  da frente da fila e, em seguida, varre a matriz de adjacência ( $g->adj[v][w]$ ) em busca de vizinhos  $w$  que ainda não foram visitados. Para cada vizinho  $w$  encontrado, o código o marca como visitado define  $v$  como seu pai ( $pai[w] = v$ ) e o insere no fim da fila. Esse processo assegura que o algoritmo explore todos os nós a uma distância  $k$  antes de explorar

### a) Ordenação topológica:

O que fizemos é uma ordenação topológica (reversa) de um DAG, ela é uma ordenação dos vértices do DAG onde um vértice que aparece na posição  $i$ , tem arcos apenas para vértices em  $\{0, 1, \dots, i-1\}$ , isto é, podemos realizar as tarefas na ordem dada.

Para encontrar a ordenação topológica, considere um vértice  $u$  do DAG:

- Todo  $v$  tal que  $(u, v)$  é arco deve aparecer antes de  $u$ ;
- Todo  $w$  tal que  $(v, w)$  é arco deve aparecer antes de  $v$ ;
- E assim por diante;

Devemos considerar todos  $w$  para os quais existe caminho de  $u$  para  $w$  antes de considerar  $u$ . Para encontrar todos tal que existe caminho de  $u$  para  $w$  devemos usar uma busca em profundidade.

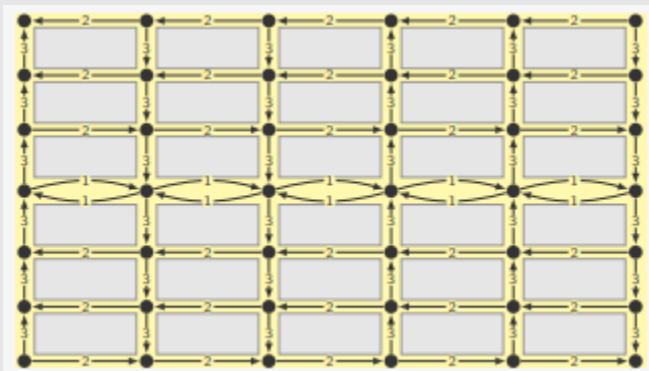
```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
9     printf("\n");
10 }
```

```
1 void visita_rec(p_grafo g, int *visitado, int v) {
2     p_no t;
3     visitado[v] = 1;
4     for (t = g->adj[v]; t != NULL; t = t->prox)
5         if (!visitado[t->v])
6             visita_rec(g, visitado, t->v);
7     printf("%d ", v);
8 }
```

(printf): um vértice só é impresso após todas as chamadas recursivas para seus descendentes (os vértices alcançáveis a partir dele) terem sido concluídas. Isso significa que o algoritmo imprime os vértices na ordem de "finalização" da busca, o que resulta em uma ordenação topológica inversa (ou seja, uma lista onde cada tarefa só aparece depois que todas as tarefas que dependem dela já foram listadas).

### b) Como encontrar o menor caminho para ir de A para B:



A função ordenação\_topologica serve como a "piloto" da busca, inicializando um vetor de visitado e garantindo que, mesmo que o grafo seja desconexo, todos os seus componentes sejam explorados ao iterar por todos os vértices  $s$ . O núcleo da lógica está na função visita\_rec: ela marca um vértice  $v$  como visitado e, em seguida, chama a si mesma recursivamente para todos os vizinhos de  $v$  que ainda não foram visitados. O comportamento-chave está na linha 7

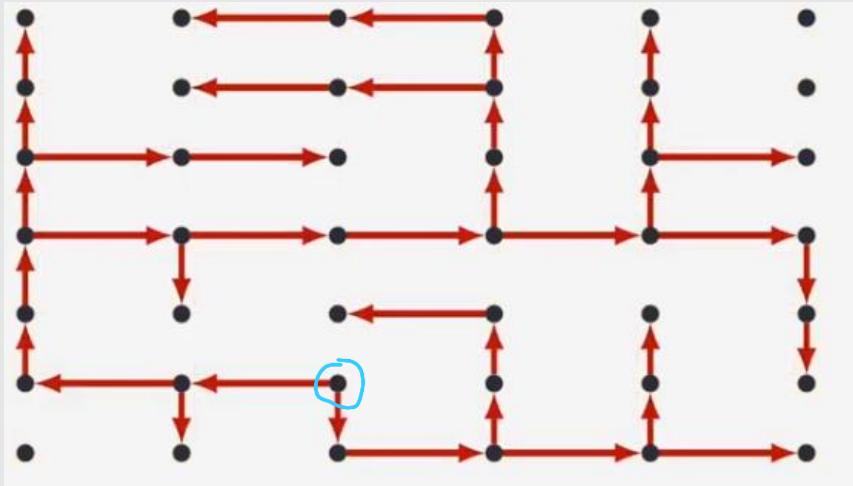
Modelamos um dígrafo com pesos nos arcos:

- Um vértice em cada cruzamento;
- Um arco entre vértices consecutivos;
- O peso do arco  $(u, v)$  é o tempo de viagem de  $u$  para  $v$ ;

Não queremos mais o caminho com o menor número de arestas como o BFS faz, agora cada aresta tem um peso e queremos encontrar o caminho de A até B com a soma total dos pesos das arestas mínima.

Para o nosso algoritmo teremos pesos não negativos (caso houvesse um ciclo negativo o tempo mínimo seria menos infinito).

Normalmente não queremos apenas o caminho mínimo de A para apenas um nó B. Queremos o caminho mínimo de A para todos outros nós que ele consegue alcançar e é aí que surge a árvore de caminhos.



Dessa forma, ao rodar o algoritmo a partir do vértice em azul ao lado, teríamos essa árvore indicada pelas flechas vermelhas.

### c) Algoritmo de Dijkstra:

Seguindo as ideias do algoritmo acima, temos um conjunto de vértices que ainda não entraram na árvore de caminhos mínimos. Alguns desses são vizinhos diretos dos vértices que já estão na árvore (como os nós vermelhos e azuis do slide anterior).

Esse conjunto de "vizinhos na fronteira" é o que chamamos de franja. O algoritmo funciona da seguinte forma:

1. A Escolha: A cada passo, o algoritmo inspeciona *todos* os vértices na franja e seleciona aquele(v) que possui a menor distância acumulada desde o vértice de origem (A).
2. Inclusão na Árvore: Esse vértice v é "finalizado" e adicionado à árvore de caminhos mínimos (tornando-se parte do conjunto vermelho). A sua distância é agora considerada a menor distância absoluta possível desde A.
3. Atualização de Vizinhos: Imediatamente, o algoritmo analisa todos os vizinhos de v (vamos chamá-los de w). Para cada vizinho w, ele calcula um novo caminho potencial: custo (A até v) + custo (v até w).
  - o Se esse novo custo for *menor* do que o custo que tínhamos registrado para w anteriormente, nós atualizamos a distância de w.
  - o É assim que o algoritmo "descobre" caminhos melhores à medida que explora o grafo.

4. Repetição: O algoritmo repete os passos 1, 2 e 3, sempre pegando o próximo vértice mais próximo na franja, até que o vértice de destino (B) seja escolhido, ou até que a franja fique vazia (o que significa que encontramos o caminho para todos os vértices alcançáveis).

```

1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {           Tempo: O(|E|lg|V|)
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
14            for (t = g->adj[v]; t != NULL; t = t->prox)
15                if (prioridade(h, v)+t->peso < prioridade(h, t->v)) {
16                    diminuiprioridade(h,t->v,prioridade(h,v)+t->peso);
17                    pai[t->v] = v;
18                }
19    }
20    return pai;
21 }
```

Este código implementa o Algoritmo de Dijkstra usando uma fila de prioridade (min-heap) para encontrar eficientemente os caminhos de menor custo a partir de um vértice de origem s. A função começa inicializando as estruturas de dados: um vetor pai (para armazenar a árvore de caminhos resultante) e

a própria fila de prioridade h. Em seguida, todos os vértices do grafo são inseridos na fila com uma prioridade (distância) "infinita" (INT\_MAX), e o vértice de origem s tem sua prioridade ajustada para 0, garantindo que ele seja o primeiro a ser processado.

O coração do algoritmo é o loop while que executa enquanto a fila de prioridade não estiver vazia. A cada iteração, ele aplica sua escolha gulosa fundamental: a função `extrai_minimo(h)` remove o vértice v da fila que possui a menor distância *conhecida* da origem. A partir desse momento, a distância até v é considerada *finalizada* e correta, e ele é "adicionado" à árvore de caminhos mínimos.

Uma vez que um vértice v é finalizado, o algoritmo entra na etapa de relaxamento (relaxation). Ele percorre todos os vizinhos de v (os vértices t->v alcançáveis a partir dele) e verifica se é possível *melhorar* o caminho até eles. A condição if (linha 15) checa se a distância finalizada de v somada ao peso da aresta (v, t->v) é menor que a distância atualmente registrada para t->v. Se for, o algoritmo encontrou um caminho mais curto para t->v: ele atualiza a distância de t->v na fila usando `diminuiprioridade` e define v como o novo pai de t->v.

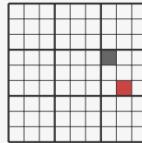
Quando o loop while termina, a fila está vazia, o que significa que todos os vértices alcançáveis foram finalizados. A função então retorna o vetor pai. Este vetor é a representação da árvore de caminhos mínimos, pois para qualquer vértice x, pode-se encontrar o caminho de volta à origem s seguindo a cadeia de predecessores (`pai[x], pai[pai[x]], ...`).

## **22. Backtracking:**

O Backtracking é uma técnica utilizada para resolver problemas de forma recursiva, onde construímos a resposta passo a passo. Começamos com uma solução parcial vazia e vamos adicionando elementos, um de cada vez. O grande "segredo" deste método é saber lidar com decisões erradas: se a qualquer momento a solução parcial deixar de ser válida ou se não for possível adicionar novos elementos, nós retrocedemos (fazemos o *backtrack*). Isto significa remover o último elemento adicionado e tentar uma decisão ou caminho diferente, repetindo este ciclo até encontrar a solução completa.

### **a) Sudoku:**

```
1 int pode_inserir(int m[9][9], int l, int c, int v) {
2     int i, j, cel_1, cel_c;
3     for (i = 0; i < 9; i++)
4         if (m[i][l] == v) // aparece na linha l?
5             return 0;
6     for (i = 0; i < 9; i++)
7         if (m[i][c] == v) // aparece na coluna c?
8             return 0;
9
10    cel_1 = 3 * (l / 3);
11    cel_c = 3 * (c / 3);
12    for (i = cel_1; i < cel_1 + 3; i++)
13        for (j = cel_c; j < cel_c + 3; j++)
14            if (m[i][j] == v) // aparece na célula?
15                return 0;
16    return 1;
17 }
```



contrário, avisa que é proibido (retorna 0).

Essa função serve para validar uma jogada no Sudoku antes de a escrevermos definitivamente. Ela verifica se o número que queremos inserir viola alguma das três regras fundamentais do jogo: se já existe na mesma linha, na mesma coluna ou no mesmo bloco 3x3. Se o número não violar nenhuma regra, a função autoriza a inserção (retorna 1); caso

```
1 int sudoku(int m[9][9]) {
2     int i, j, fixo[9][9];
3     for (i = 0; i < 9; i++)
4         for (j = 0; j < 9; j++)
5             fixo[i][j] = m[i][j]; /* diferente de zero é verdadeiro */
6     return sudokuR(m, fixo, 0, 0);
7 }
8
9 void proxima_posicao(int l, int c, int *nl, int *nc) {
10    if (c < 8) {
11        *nl = 1;
12        *nc = c + 1;
13    } else {
14        *nl = l + 1;
15        *nc = 0;
16    }
17 }
```

Essas funções servem para preparar o tabuleiro e controlar a navegação. A função *sudoku* faz uma cópia de segurança dos números iniciais (as "pistas") para a matriz *fixo*, garantindo que o programa sabe quais as células que não pode apagar, e inicia a resolução. Já a função *proxima\_posicao* age como um cursor automático:

ela calcula as coordenadas da próxima casa a visitar, avançando para a direita e, quando a linha acaba, saltando para o início da linha de baixo.

Esta é a função principal que orquestra todo o processo recursivo. Primeiro, ela verifica se chegámos ao fim do tabuleiro (se a linha é 9), o que significa que resolvemos o puzzle. Se a célula atual for um número fixo, ela avança automaticamente. Se estiver vazia, ela entra num ciclo para testar números de 1 a 9. Quando encontra um válido, coloca-o na matriz e chama-se a si própria para tentar o próximo passo. A "magia" do

```
1 int sudokuR(int m[9][9], int fixo[9][9], int l, int c) {
2     int v, nl, nc;
3     if (l == 9) {
4         imprime_sudoku(m);
5         return 1;
6     }
7     proxima_posicao(l, c, &nl, &nc);
8     if (fixo[l][c])
9         return sudokuR(m, fixo, nl, nc);
10    for (v = 1; v <= 9; v++) {
11        if (pode_inserir(m, l, c, v)) {
12            m[l][c] = v;
13            if (sudokuR(m, fixo, nl, nc))
14                return 1;
15        }
16    }
17    m[l][c] = 0;
18    return 0;
19 }
```

Backtracking acontece na linha 17: se o caminho escolhido der errado lá na frente, ela apaga o número (coloca 0) e retorna falso, permitindo voltar atrás e tentar uma nova opção.

A grande vantagem da eficiência do Backtracking é não perder tempo com caminhos impossíveis. Ao contrário da Força Bruta, que tenta todas as combinações cegamente, o Backtracking "corta" as opções assim que percebe que uma solução parcial é inválida. Para que funcione bem, a regra de validação — no nosso caso, a função `pode_inserir` — tem de ser rápida e inteligente: quanto mais cedo detetarmos que um caminho não tem futuro, menos trabalho o computador tem e mais rápido chegamos à resposta final.