

# Estrutura de Dados para Armazenamento de Informações

## Vetores em C

- Vetores não ordenados
- Vetores ordenados

# Conceitos Iniciais: Alocação de Memória

- Quando criamos uma variável em um programa C, nós estamos **reservando** (ou **alocando**) **um espaço na memória RAM** para guardar o valor dessa variável.

Exemplo: `int idade = 20;` O compilador reserva **4 bytes** na memória RAM para armazenar o valor 20.

Tipo de dado	Espaço ocupado (em geral)	Exemplo de uso
char	1 byte	Letras e caracteres
int	4 bytes	Números inteiros pequenos/médios
float	4 bytes	Números com casas decimais
double	8 bytes	Números decimais de alta precisão

## Importância da Alocação.

- Permite que o programa saiba **onde encontrar e como manipular** os dados na memória.
- Evita que duas variáveis **ocupem o mesmo lugar**, o que poderia gerar **erros e comportamentos imprevisíveis**. O que permite que o programa controle e manipule os dados de forma eficiente.

# Como um programa é alocado em memória?

- Quando você executa um programa em C, o sistema operacional **carrega** o programa na memória RAM, dividindo-o em **várias regiões específicas**.

Região da memória	O que contém	Exemplo
<b>Código (Text Segment)</b>	As instruções do programa (funções, código executável)	Código das funções main(), printf()
<b>Dados Estáticos (Data Segment)</b>	Variáveis globais e estáticas <b>inicializadas</b>	int contador = 0;
<b>BSS (Block Started by Symbol)</b>	Variáveis globais e estáticas <b>não inicializadas</b>	static int valor;
<b>Heap</b>	Área para alocação dinâmica	Vetores criados dinamicamente
<b>Stack (Pilha)</b>	Área para variáveis locais e chamadas de funções	Variáveis dentro de funções

# Exemplos

## 1. Text Segment (Segmento de Código)

### O que é?

- Armazena o **código do programa compilado**: funções, comandos, lógicas.
- Só de **leitura** (para evitar alteração accidental do código).

### Exemplo:

```
int soma(int a, int b) {  
    return a + b;  
}
```

Essa função soma fica armazenada no **Text Segment**.

## 3. BSS Segment (Dados Não Inicializados)

### O que é?

- Guarda **variáveis globais ou estáticas que não foram inicializadas** explicitamente.
- O sistema zera essa área automaticamente (coloca 0).

### Exemplo:

```
static int total; // Sem valor inicial  
int contador; // Variável global sem inicializar  
Essas variáveis ficam na BSS, esperando serem usadas.
```

## 2. Data Segment (Dados Inicializados)

### O que é?

- Guarda **variáveis globais ou estáticas que foram inicializadas** com algum valor antes da execução.

### Exemplo:

```
int global = 10; // Variável global inicializada static int  
contagem = 5; // Variável estática inicializada  
Essas variáveis são armazenadas no Data Segment já com seus valores (10 e 5).
```

## 4. Heap (Área de Alocação Dinâmica)

### O que é?

- Área usada para **reservar memória em tempo de execução** — não se sabe o tamanho na hora da compilação.

- Usada para malloc(), calloc(), realloc().

### Exemplo:

```
int *p = malloc(10 * sizeof(int)); // Cria espaço para 10 inteiros na Heap
```

Sempre que você usar **alocação dinâmica**, o espaço é pego da **Heap**.

Atenção: o programador deve **liberar a memória** depois (free(p));!

# Exemplos:

## 5. Stack (Pilha de Execução)

O que é?

- Área para:

- Variáveis locais** dentro de funções
- Parâmetros de função**
- Controle de chamadas de função** (quem chamou quem)
- Funciona como uma **pilha** (último a entrar, primeiro a sair - LIFO).

**Exemplo:**

```
void funcao() { int a = 5; // Variável local (fica na Stack) }
```

Quando funcao() é chamada:

- Um **bloco** (chamado de **frame de ativação**) é criado na Stack para guardar a.
- Quando funcao() termina, esse espaço é automaticamente **liberado**.

# Resumão:

Região	Guarda o quê?	Criada quando?
Text Segment	Código do programa	Compilação
Data Segment	Variáveis globais <b>inicializadas</b>	Compilação
BSS Segment	Variáveis globais <b>não inicializadas</b>	Compilação
Heap	Memória dinâmica (malloc)	Em tempo de execução
Stack	Variáveis locais, funções	Em tempo de execução

# Como variáveis são alocadas em memória?

Tipo de variável	Onde fica?	Exemplo
Variáveis locais	Na <b>Stack</b>	Declaradas dentro de funções (int a;)
Variáveis globais e estáticas	No <b>Data Segment ou BSS</b>	Declaradas fora de funções
Variáveis dinâmicas	Na <b>Heap</b>	Usam malloc, calloc, realloc

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

int global = 10;           // Alocado no segmento de dados

int main() {
    int local = 5;         // Alocado na Stack

    int *dinamica = malloc(sizeof(int)); // Alocado na Heap
    *dinamica = 20;

    printf("%d %d %d\n", global, local, *dinamica);

    free(dinamica); // Libera memória alocada dinamicamente
    return 0;
}
```

# Alocação é estática ou dinâmica?

- Depende do tipo de variável e como ela é criada no programa

Tipo de Alocação	Como é feita	Quando é conhecida	Exemplo C
Estática	Em tempo de compilação	Antes da execução	int idade = 20;
Dinâmica	Em tempo de execução	Durante a execução	int *p = malloc(10*sizeof(int));

**É melhor alocar toda a memória de uma vez ou ir alocando dinamicamente conforme necessário?**

Resposta : **Depende da situação e do tipo de programa** que você está desenvolvendo.

Situação	Melhor escolha	Por quê?
Programas pequenos/fixos	Estática	Simples, sem complexidade desnecessária
Programas grandes, dados variáveis	Dinâmica	Flexibilidade e economia de memória

# Conceito Inicial: Linearidade x Não Linearidade

- As estruturas de dados podem ser divididas em dois grandes grupos:

Tipo

**Lineares**

**Não lineares**

Conceito Resumido

Os elementos são organizados **um após o outro**.

Os elementos são organizados em **ramificações, hierarquias ou relações complexas**.

# Estruturas de Dados Lineares

- Os elementos estão organizados de forma **sequencial**, como uma fila ou uma pilha de livros. Há uma ordem clara do primeiro ao último elemento.

Estrutura

**Vetores / Listas**

**Pilhas (Stacks)**

**Filas (Queues)**

**Listas Ligadas**

Características principais

Acesso por índice, fixo ou dinâmico

LIFO (último a entrar, primeiro a sair)

FIFO (primeiro a entrar, primeiro a sair)

Elementos ligados por ponteiros

# Estruturas de Dados Não Lineares

- Os dados são organizados em **estruturas hierárquicas ou em rede**, sem sequência única direta entre os elementos.

Estrutura

**Árvores**

**Grafos**

**Tabelas Hash**

Características principais

Dados organizados em formato de hierarquia (ex: árvore genealógica)

Elementos com múltiplas conexões (ex: redes sociais, mapas de trânsito)

Mapeamento chave-valor com busca eficiente

# Comparação Geral

Critério	Lineares	Não Lineares
Organização dos dados	Sequencial	Hierárquica ou em rede
Navegação	Simples (por índice ou ponteiro)	Mais complexa (por caminhos ou chaves)
Acesso direto	Mais comum	Menos comum (exceto hash tables)
Exemplo de uso comum	Fila de impressão, histórico	Árvore de arquivos, redes sociais

# Vetor: O que é um vetor?

- Um vetor é uma **estrutura de dados linear** que armazena **valores do mesmo tipo**, um ao lado do outro, em uma **sequência ordenada** de posições (índices).

## Analogia simples:

Imagine uma **caixa de ovos** com 12 espaços. Cada espaço tem um número (de 0 a 11). Você pode colocar e pegar um ovo em qualquer posição — **desde que saiba o número da posição**. Essa caixa é o nosso **vetor**.

As posições são os **índices**.

Os ovos são os **valores** que queremos armazenar.

# Vetores Não Ordenados

- Um **vetor não ordenado** é aquele em que **não há uma regra ou ordem específica para os elementos armazenados**. Isso significa que os dados podem estar em qualquer sequência e que **não existe garantia** de que os elementos estão organizados por valor, ordem alfabética, ou qualquer critério lógico.
- Exemplo de vetor não ordenado:

```
nomes = ["Carlos", "Ana", "João", "Beatriz"]
```

*Neste vetor, os nomes não estão ordenados alfabeticamente nem por tamanho. É simplesmente uma lista de elementos armazenados juntos.*

# Vetores Não Ordenados em C

## Conceito rápido:

Em **vetores não ordenados**, os elementos são inseridos **no final** da sequência, **sem preocupação com a ordem**.

A busca é feita **de forma linear**, verificando um por um.

A exclusão é feita **deslocando** os elementos subsequentes.

### Operação

Inserção

Pesquisa

Exclusão

### Como funciona no vetor não ordenado

Sempre no final

Linear (do começo ao fim)

Remover e deslocar os elementos para trás

# Inserção em Vetor Não Ordenado

```
#include <stdio.h> // Biblioteca padrão para funções de entrada e saída

#define TAMANHO 10 // Definindo o tamanho máximo do vetor

int main() {
    int vetor[TAMANHO]; // Declarando um vetor de inteiros com espaço para 10 elementos
    int n = 0;           // Variável que guarda quantos elementos já inserimos (inicialmente vazio)

    // Inserindo valores
    vetor[n++] = 10;    // Inserimos 10 na posição 0 e incrementamos n
    vetor[n++] = 30;    // Inserimos 30 na posição 1 e incrementamos n
    vetor[n++] = 20;    // Inserimos 20 na posição 2 e incrementamos n

    printf("Elementos do vetor:\n"); // Mensagem para identificar os dados que serão exibidos
    for (int i = 0; i < n; i++) { // Laço para percorrer os elementos já inseridos
        printf("Posição %d: %d\n", i, vetor[i]); // Exibe a posição e o valor correspondente
    }

    return 0; // Finaliza o programa
}
```

**Saída esperada:**  
Elementos do vetor:  
Posição 0: 10  
Posição 1: 30  
Posição 2: 20

# Pesquisa Linear em Vetor Não Ordenado

```
#include <stdio.h> // Biblioteca padrão de entrada e saída

#define TAMANHO 10 // Define o tamanho máximo do vetor

int main() {
    int vetor[TAMANHO] = {10, 30, 20}; // Inicializa o vetor com 3 valores
    int n = 3; // Indica que temos 3 elementos no vetor
    int valor = 30; // Valor que queremos buscar

    int encontrado = -1; // Variável para armazenar a posição do valor (inicialmente -1 = não encontrado)

    // Pesquisa linear: percorre o vetor do começo ao fim
    for (int i = 0; i < n; i++) {
        if (vetor[i] == valor) { // Se encontrar o valor desejado
            encontrado = i; // Guarda a posição onde encontrou
            break; // Para o laço (não precisa procurar mais)
        }
    }

    // Mostrando o resultado da pesquisa
    if (encontrado != -1) { // Se encontrou o valor
        printf("Valor %d encontrado na posição %d.\n", valor, encontrado);
    } else { // Se não encontrou
        printf("Valor %d não encontrado.\n", valor);
    }
}

return 0; // Finaliza o programa
}
```

## Saída esperada:

Valor 30 encontrado na posição 1.

# Exclusão de um Elemento no Vetor Não Ordenado

```
#include <stdio.h> // Biblioteca padrão de entrada e saída
#define TAMANHO 10 // Define o tamanho máximo do vetor
int main() {
    int vetor[TAMANHO] = {10, 30, 20}; // Vetor inicializado com 3 valores
    int n = 3; // Número de elementos no vetor
    int valor = 30; // Valor que queremos excluir

    int posicao = -1; // Variável para armazenar a posição do valor a excluir (inicialmente -1 = não encontrado)
    // Pesquisa para encontrar a posição do valor
    for (int i = 0; i < n; i++) {
        if (vetor[i] == valor) { // Se o valor encontrado for igual ao procurado
            posicao = i; // Armazena a posição
            break; // Para o laço
        }
    }
    // Verifica se encontrou o valor para excluir
    if (posicao != -1) {
        // Desloca os elementos após a posição encontrada uma casa para a esquerda
        for (int i = posicao; i < n - 1; i++) {
            vetor[i] = vetor[i + 1]; // Move o próximo valor para a posição atual
        }
        n--; // Decrementa o número de elementos no vetor
        printf("Valor %d excluído.\n", valor); // Informa que a exclusão foi feita
    } else {
        printf("Valor %d não encontrado.\n", valor); // Informa que o valor não foi encontrado
    }
    // Exibe o vetor atualizado
    printf("Elementos do vetor após exclusão:\n");
    for (int i = 0; i < n; i++) {
        printf("Posição %d: %d\n", i, vetor[i]);
    }
    return 0; // Finaliza o programa
}
```

**Saída esperada:**  
Valor 30 excluído.  
Elementos do vetor  
após exclusão:  
Posição 0: 10  
Posição 1: 20

# Tipos de Vetores

- Existem diferentes formas de trabalhar com vetores, cada uma com características específicas, dependendo da aplicação.

Tipo de Representação	Inserção rápida	Acesso rápido (por índice)	Remoção rápida	Mantém ordenação
Array Estático (fixo)	<input checked="" type="checkbox"/> Sim (no final)	<input checked="" type="checkbox"/> Sim (por índice)	<input type="checkbox"/> Não (precisa deslocar)	<input type="checkbox"/> Não
Array Dinâmico (malloc)	<input checked="" type="checkbox"/> Sim (inicialmente)	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não (deslocamento)	<input type="checkbox"/> Não
Lista Encadeada	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não (tem que percorrer)	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não
Heap (heapq conceito)	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não (não acesso direto)	<input checked="" type="checkbox"/> Sim (menor elemento)	<input type="checkbox"/> Não (ordem parcial)

# Array Estático

## O que é:

Um vetor comum, criado com tamanho fixo definido **na hora da compilação**.

Exemplo:

```
int vetor[10];
```

- **O tamanho é fixo**: não pode aumentar nem diminuir depois que o programa roda.
- **Simples de usar**.
- **Acesso rápido** (vetor[i] diretamente).
- **Inserção**: adiciona sempre na próxima posição disponível (vetor[n++] = valor).
- **Pesquisa**: linear, percorrendo elemento por elemento.

# Array Estático

```
#include <stdio.h> // Biblioteca para entrada e saída

int main() {
    int vetor[5];      // Declara um vetor de 5 inteiros
    int n = 0;          // Variável para controlar o número de elementos inseridos

    vetor[n++] = 10;   // Insere 10 na posição 0 e incrementa n
    vetor[n++] = 40;   // Insere 40 na posição 1 e incrementa n
    vetor[n++] = 20;   // Insere 20 na posição 2 e incrementa n

    printf("Vetor não ordenado (Array Estático):\n"); // Exibe mensagem antes de mostrar o vetor

    for (int i = 0; i < n; i++) { // Percorre os elementos já inseridos
        printf("Posição %d: %d\n", i, vetor[i]); // Mostra o índice e o valor armazenado
    }

    return 0; // Encerra o programa
}
```

**Saída esperada:**  
Vetor não ordenado  
(Array Estático):  
Posição 0: 10  
Posição 1: 40  
Posição 2: 20

## Observação:

- **Tamanho fixo:** não podemos adicionar mais elementos depois.
- **Acesso direto** ao elemento pelo índice: vetor[i].

# Array Dinâmico (malloc)

O que é:

Um vetor criado usando **alocação dinâmica** (malloc) em tempo de execução.

Exemplo:

```
int *vetor;  
vetor = malloc(tamanho * sizeof(int));
```

- **Tamanho definido na execução** (não na compilação).
- **Pode ser realocado** (realloc) para crescer, se necessário.
- **É necessário liberar memória** (free) manualmente depois de usar.

# Array Dinâmico (malloc)

Um vetor criado em tempo de execução usando malloc(). Pode ser ajustado conforme o tamanho que precisamos.

```
#include <stdio.h>      // Biblioteca padrão de entrada e saída
#include <stdlib.h>      // Biblioteca para alocação dinâmica de memória

int main() {
    int *vetor;          // Declara um ponteiro para inteiro
    int n = 0;            // Número de elementos atualmente no vetor
    int tamanho = 5;      // Tamanho máximo inicial para o vetor

    vetor = malloc(tamanho * sizeof(int)); // Aloca memória para 5 inteiros

    vetor[n++] = 15;      // Insere 15 na posição 0
    vetor[n++] = 25;      // Insere 25 na posição 1
    vetor[n++] = 5;       // Insere 5 na posição 2

    printf("Vetor não ordenado (Array Dinâmico):\n"); // Mensagem inicial

    for (int i = 0; i < n; i++) { // Percorre os elementos inseridos
        printf("Posição %d: %d\n", i, vetor[i]); // Exibe a posição e o valor
    }

    free(vetor); // Libera a memória alocada dinamicamente

    return 0; // Finaliza o programa
}
```

## Saída esperada:

Vetor não ordenado  
(Array Dinâmico):

Posição 0: 15

Posição 1: 25

Posição 2: 5

# Lista Encadeada

(Cada elemento aponta para o próximo.)

O que é:

Uma estrutura onde cada elemento aponta para o próximo elemento.

Não existe um vetor "contínuo": cada nó armazena um valor e o endereço do próximo.

Exemplo:

```
typedef struct No {  
    int valor;  
    struct No *prox; // Aponta para o próximo nó  
} No;
```

- Permite **inserir e remover facilmente** sem deslocar vários dados.
- **Inserção**: criar um novo nó e ligar na lista.
- **Pesquisa**: precisa percorrer os nós manualmente.
- **Acesso aleatório**: impossível (diferente de vetor).

# Lista Encadeada

(Cada elemento aponta para o próximo.)

```
#include <stdio.h> // Biblioteca padrão de entrada e saída
#include <stdlib.h> // Biblioteca para funções de alocação dinâmica
// Definição da estrutura do nó
typedef struct No {
    int valor; // Valor armazenado no nó
    struct No *proxímo; // Ponteiro para o próximo nó
} No;
// Função para inserir um novo nó no início da lista
void inserir_início(No **início, int valor) {
    No *novo = malloc(sizeof(No)); // Aloca memória para o novo nó
    novo->valor = valor; // Define o valor do novo nó
    novo->proxímo = *início; // Faz o novo nó apontar para o antigo primeiro nó
    *início = novo; // Atualiza o início da lista
}
// Função para imprimir a lista
void imprimir(No *início) {
    No *atual = início; // Começa pelo primeiro nó
    printf("Lista: ");
    while (atual != NULL) { // Enquanto houver nós
        printf("%d -> ", atual->valor); // Imprime o valor
        atual = atual->proxímo; // Avança para o próximo nó
    }
    printf("NULL\n"); // Fim da Lista
}
int main() {
    No *lista = NULL; // Inicializa a lista como vazia
    // Inserções
    inserir_início(&lista, 30);
    inserir_início(&lista, 20);
    inserir_início(&lista, 10);
    imprimir(lista); // Imprime a lista encadeada
    return 0;
}
```

## Saída esperada:

Lista: 10 -> 20 -> 30 -> NULL

Cada elemento (nó) aponta para o próximo, formando uma sequência.

## Características:

- Inserção/remover é fácil e rápido (não precisa deslocar).
- Acesso a elementos é lento (precisa percorrer de um em um).
- Não mantém ordenação naturalmente

# Heap (heapq conceito)

## Heap (heapq conceito)

### O que é?

- Heap é uma estrutura que **organiza os dados parcialmente**:
  - O **menor elemento** (ou maior) fica no topo.
  - Os outros elementos **não estão necessariamente ordenados**.

### Pergunta:

- É vetor **não ordenado**?

**Sim!** Porque o vetor inteiro **não é ordenado**, apenas o topo é organizado.

### Complicação:

- Heap exige conhecimento de:
  - Organização de árvore binária
  - Índices de filhos e pais

Voltaremos  
no tema qdo  
falarmos de  
Arvores

# Resumo Comparativo Rápido

## Situação

Poucos dados fixos

Dados de tamanho variável

Muitas inserções e remoções rápidas

Trabalhar com prioridade (menor/maior primeiro)

## Melhor Tipo de Vetor em C

Array Estático

Array Dinâmico (malloc/realloc)

Lista Encadeada

Heap

# Vetores Ordenados

- Um **vetor ordenado** é uma estrutura de dados linear em que os elementos são armazenados em **ordem crescente ou decrescente**, conforme um critério definido (geralmente valor numérico ou ordem alfabética).
- Exemplo de vetor ordenado:

```
idades = [18, 21, 23, 25, 30, 34] # Ordenado em ordem crescente
```

*Diferente de um vetor comum (não ordenado), onde os valores podem estar em qualquer ordem.*

# Diferença entre Vetor Ordenado e Não Ordenado

Característica	Vetor Não Ordenado	Vetor Ordenado
Inserção	Rápida (no final)	Lenta (precisa encontrar posição e deslocar)
Pesquisa Linear	Sim (obrigatória)	Sim, mas otimizada
Pesquisa Binária	Não é possível	<input checked="" type="checkbox"/> Sim, é possível
Exclusão	Exige pesquisa e deslocamento	Exige pesquisa e deslocamento
Manutenção da ordem	 Não existe	<input checked="" type="checkbox"/> Sempre mantido
Facilidade de implementação	Muito fácil	Um pouco mais complexo (deslocamentos)
Eficiência em grandes volumes	Baixa	Alta (por causa da pesquisa binária)

# Vetores Ordenados

A ordenação torna **a busca mais eficiente e os dados mais organizados**.

Em vetores ordenados, é possível utilizar **busca binária**, que encontra elementos muito mais rápido do que a busca linear.

## Vantagens dos Vetores Ordenados

- Dados prontos para **relatórios, visualização ou ordenação por faixas**.
- Evita duplicatas de forma mais simples (verificando a posição correta antes de inserir).
- Facilita integração com **algoritmos de classificação** ou estruturas como árvores平衡adas.

Vantagem	Explicação
<b>Pesquisa mais rápida</b>	Pode usar <b>Busca Binária</b> , que é muito mais eficiente ( $O(\log n)$ comparado a $O(n)$ da busca linear).
<b>Facilita buscas sucessivas</b>	Se precisar fazer muitas consultas, um vetor ordenado economiza muito tempo.
<b>Facilita operações futuras</b>	Operações como <b>interseção, união, remoção duplicada</b> ficam muito mais rápidas e simples.
<b>Estrutura base para outras estruturas</b>	Muitas estruturas mais avançadas (como árvores binárias, tabelas de busca) usam vetores ordenados como base.
<b>Visualização fácil</b>	Para quem lê ou apresenta dados, ver uma sequência ordenada é muito mais intuitivo.

# Inserção em Vetor Ordenado

```
#include <stdio.h> // Biblioteca padrão de entrada e saída

#define TAM 10 // Tamanho máximo do vetor

int main() {
    int vetor[TAM] = {10, 20, 30, 50}; // Vetor já preenchido com 4 elementos ordenados
    int n = 4; // Número atual de elementos no vetor
    int novo = 25; // Valor que queremos inserir
    int i = n - 1; // Começamos pelo último elemento preenchido

    // Etapa 1: Encontrar onde o novo valor deve ser inserido
    while (i >= 0 && vetor[i] > novo) { // Enquanto o valor atual for maior que o novo
        vetor[i + 1] = vetor[i]; // Empurra o valor para a direita
        i--; // Move para o elemento anterior
    }

    vetor[i + 1] = novo; // Insere o novo valor na posição correta
    n++; // Atualiza o número de elementos

    // Etapa 2: Mostrar o vetor atualizado
    printf("Vetor após inserção: ");
    for (i = 0; i < n; i++) { // Percorre o vetor atualizado
        printf("%d ", vetor[i]); // Imprime cada elemento
    }
    printf("\n");

    return 0; // Finaliza o programa
}
```

- Procuramos **onde** o novo valor deve entrar;
- **Deslocamos os outros valores para a direita** (se necessário);
- Inserimos o valor **na posição correta**.

## O que aconteceu passo a passo:

- 1 Começamos com o vetor [10, 20, 30, 50].
- 2 Queremos inserir o valor 25.
- 3 Comparamos de trás para frente:
  - $50 > 25 \rightarrow$  empurramos 50 para a frente.
  - $30 > 25 \rightarrow$  empurramos 30 para a frente.
  - $20 < 25 \rightarrow$  paramos aqui.
- 4 Inserimos 25 logo depois do 20.
- 5 O vetor ficou [10, 20, 25, 30, 50].
- 6 Imprimimos o vetor atualizado.

# Pesquisa Linear em Vetor Ordenado

Ela varre o vetor do começo até o fim, comparando cada valor com o que estamos procurando.

```
#include <stdio.h>      // Biblioteca padrão de entrada e saída
#define TAM 6            // Define o tamanho do vetor
int main() {
    int vetor[TAM] = {5, 10, 15, 20, 30, 50}; // Vetor já ordenado
    int valor = 20; // Valor que queremos procurar
    int i; // Variável de controle do laço
    int encontrado = -1; // Variável para guardar a posição encontrada (-1 = não encontrado)

    // Vamos percorrer o vetor
    for (i = 0; i < TAM; i++) {
        printf("Comparando com posição %d: %d\n", i, vetor[i]); // Mostra a posição atual e o valor

        if (vetor[i] == valor) { // Se encontramos o valor
            encontrado = i; // Armazenamos a posição
            printf("Valor %d encontrado na posição %d\n", valor, i);
            break; // Paramos a busca
        } else if (vetor[i] > valor) { // Se o valor atual é maior que o procurado
            printf("Parando: valor %d não está no vetor.\n", valor);
            break; // Podemos parar porque o vetor é ordenado
        }
    }
    // Se não encontrou o valor
    if (encontrado == -1) {
        printf("Valor %d não encontrado no vetor.\n", valor);
    }
    return 0; // Finaliza o programa
}
```

## Explicação passo a passo:

- 1.O vetor é [5, 10, 15, 20, 30]
- 2.O valor procurado é 20
- 3.O laço começa:
  - **Posição 0:**  $5 \neq 20 \rightarrow$  continua
  - **Posição 1:**  $10 \neq 20 \rightarrow$  continua
  - **Posição 2:**  $15 \neq 20 \rightarrow$  continua
  - **Posição 3:**  $20 == 20 \rightarrow$  achou!
- 4.A função retorna 3 (a posição onde encontrou o valor)

# Exclusão em Vetor Ordenado

- **Encontrar a posição do valor a ser removido** (com busca linear ou binária);
- **Remover o valor**, deslocando os elementos seguintes para preencher o "buraco" deixado.

## Explicando passo a passo:

- 1.Começamos com o vetor: [5, 10, 20, 25, 30]
- 2.Queremos excluir 20
- 3.O laço percorre o vetor:
  - Posição 0: 5 → continua
  - Posição 1: 10 → continua
  - Posição 2: **achou 20**
- 4.Depois de Achar:
  - Remove o elemento da posição 2
  - Todos os elementos à direita (25, 30) **são deslocados uma posição à esquerda**
- 5.Resultado: [5, 10, 25, 30]

```
#include <stdio.h> // Biblioteca padrão de entrada e saída
#define TAM 10 // Tamanho máximo do vetor
int main() {
    int vetor[TAM] = {5, 10, 20, 25, 30}; // Vetor ordenado
    int n = 5; // Número atual de elementos
    int valor = 20; // Valor que queremos excluir
    int i, j; // Variáveis para controle dos laços
    int encontrado = -1; // Guarda a posição onde o valor foi encontrado
    // Etapa 1: Procurar o valor no vetor
    for (i = 0; i < n; i++) {
        printf("Comparando com posição %d: %d\n", i, vetor[i]); // Mostra cada comparação
        if (vetor[i] == valor) { // Se encontramos o valor
            encontrado = i; // Guardamos a posição
            printf("Valor %d encontrado na posição %d\n", valor, i);
            break; // Paramos a busca
        } else if (vetor[i] > valor) { // Se o valor atual for maior, não precisa continuar
            printf("Valor não encontrado (interrompido pela ordenação).\n");
            break;
        }
    }
    // Etapa 2: Se encontramos, removemos o valor
    if (encontrado != -1) {
        // Deslocar todos os elementos seguintes para a esquerda
        for (j = encontrado; j < n - 1; j++) {
            vetor[j] = vetor[j + 1]; // Puxa o elemento seguinte para a posição atual
        }
        n--; // Decrementa o número de elementos
        printf("Valor %d removido do vetor.\n", valor);
    } else {
        printf("Valor %d não encontrado no vetor.\n", valor);
    }
    // Mostrar o vetor atualizado
    printf("Vetor após exclusão: ");
    for (i = 0; i < n; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
    return 0; // Finaliza o programa
}
```

# Pesquisa Binária (Busca Rápida)

- A **pesquisa binária** é uma técnica usada **exclusivamente em vetores ordenados**.  
Ela **não percorre todos os elementos** — ela **divide o vetor pela metade** repetidamente até encontrar (ou não) o valor desejado.
- Se o valor procurado for menor que o valor do meio → busca à esquerda. Se for maior → busca à direita.

## Analogia simples:

Imagine procurar uma palavra no dicionário:

- Você **não começa na primeira página**.
- Você **abre no meio**, depois decide se vai para frente ou para trás.
- Isso é pesquisa binária!

# Pesquisa Binária (Busca Rápida)

```
#include <stdio.h> // Biblioteca padrão para entrada e saída
#define TAM 6 // Tamanho do vetor
int main() {
    int vetor[TAM] = {5, 10, 15, 20, 25, 30}; // Vetor já ordenado
    int valor = 20; // Valor que queremos encontrar
    int inicio = 0; // Posição inicial da busca
    int fim = TAM - 1; // Posição final da busca (último índice)
    int meio; // Variável para calcular o meio
    int encontrado = -1; // Se encontrar, guardamos a posição; começa com -1

    // Enquanto ainda tiver elementos para buscar
    while (inicio <= fim) {
        meio = (inicio + fim) / 2; // Calcula o meio do vetor
        printf("Verificando posição %d: %d\n", meio, vetor[meio]); // Mostra a posição e valor atual
        if (vetor[meio] == valor) { // Se achou o valor
            encontrado = meio; // Guarda a posição
            printf("Valor %d encontrado na posição %d.\n", valor, meio);
            break; // Para o laço (não precisa mais buscar)
        } else if (vetor[meio] < valor) { // Se o valor no meio é menor
            inicio = meio + 1; // Continua buscando na metade direita
        } else { // Se o valor no meio é maior
            fim = meio - 1; // Continua buscando na metade esquerda
        }
    }
    // Depois da busca
    if (encontrado == -1) { // Se não encontrou
        printf("Valor %d não encontrado no vetor.\n", valor);
    }
    return 0; // Finaliza o programa
}
```

## Explicando passo a passo:

### Vetor inicial:

[5, 10, 15, 20, 25, 30]

### Valor procurado: 20

#### 1. Início = 0, Fim = 5

- Meio =  $(0 + 5) // 2 = 2 \rightarrow \text{vetor}[2] = 15$
- $15 < 20 \rightarrow$  vamos buscar na **metade direita** (início = 3)

#### 2. Início = 3, Fim = 5

- Meio =  $(3 + 5) // 2 = 4 \rightarrow \text{vetor}[4] = 25$
- $25 > 20 \rightarrow$  vamos buscar na **metade esquerda** (fim = 3)

#### 3. Início = 3, Fim = 3

- Meio =  $(3 + 3) // 2 = 3 \rightarrow \text{vetor}[3] = 20$
- Achamos!

# Pesquisa Binária (Busca Rápida)

🔍 Rastreamento visto por tabela:

Etapa	Início	Fim	Meio	vetor[meio]	Ação
1	0	5	2	15	$20 > 15 \rightarrow$ vai pra direita
2	3	5	4	25	$20 < 25 \rightarrow$ vai pra esquerda
3	3	3	3	20	✅ Encontrado na posição 3

🔍 comparação de métodos

Método	Passos máximos em vetor de 1000 itens
Pesquisa Linear	Até 1000 comparações
Pesquisa Binária	Máximo de 10 comparações ( $\log_2(1000)$ )

# ordenação de Vetores(ex: Bubble Sort, Selection Sort, QuickSort...).

- Em **Python**, existem várias **estruturas prontas** (list, bisect, SortedList, heapq, numpy.array) que **automaticamente** ajudam a lidar com ordenação de listas e vetores.
- C, por ser uma linguagem **mais básica e de baixo nível, não tem essas estruturas prontas**.

Em C:

- Vetor (array) é **apenas um bloco de memória**.
- Toda **ordenação** precisa ser feita **manualmente** (ex: Bubble Sort, Selection Sort, QuickSort...).
- **Não existe**:
  - "Vetor que se auto-ordena",
  - "Inserção automática ordenada",
  - "Fila de prioridade nativa".

# Bubble Sort

## Conceito:

- O **Bubble Sort** é um dos algoritmos de ordenação **mais simples**.
- Ele funciona assim:
  - Percorre o vetor várias vezes.
  - **Compara pares de elementos vizinhos**.
  - **Troca** os elementos se estiverem fora de ordem.
  - A cada passada, o **maior elemento "sobe para o final** (como uma bolha de ar subindo na água).
-  Fácil de entender.
-  Lento para vetores grandes.

### Quando usar?

- Quando você só precisa da lista ordenada de vez em quando.

- Ideal para **iniciantes** e ensino básico.

### Limitações:

- A lista não permanece ordenada após cada inserção.
- Você precisa **ordenar manualmente** sempre que quiser usá-la ordenada.

# Listas (Vetores) Bubble Sort

- Em C, vetores (arrays) **não têm função pronta** para ordenar.
- Para ordenar, precisamos **programar a ordenação manualmente** usando um algoritmo, como o **Bubble Sort**.

```
#include <stdio.h> // Inclui a biblioteca padrão de entrada e saída
#define TAM 5 // Define o tamanho do vetor como 5
int main() { // Função principal do programa
    int numeros[TAM] = {50, 10, 30, 20, 40}; // Declara o vetor com 5 números desordenados
    int i, j, temp; // Declara variáveis auxiliares para os laços e para a troca de valores
    // Algoritmo Bubble Sort: ordena o vetor em ordem crescente
    for (i = 0; i < TAM - 1; i++) { // Laço externo: controla quantas vezes vamos passar no vetor
        for (j = 0; j < TAM - 1 - i; j++) { // Laço interno: compara elementos adjacentes
            if (numeros[j] > numeros[j + 1]) { // Se o número atual for maior que o próximo
                temp = numeros[j]; // Guarda o número atual em uma variável temporária
                numeros[j] = numeros[j + 1]; // Move o próximo número para a posição atual
                numeros[j + 1] = temp; // Coloca o número guardado na posição seguinte
            }
        }
    }
    // Imprime o vetor já ordenado
    printf("Vetor ordenado: "); // Mostra o texto na tela
    for (i = 0; i < TAM; i++) { // Laço para percorrer o vetor
        printf("%d ", numeros[i]); // Imprime cada número seguido de espaço
    }
    printf("\n"); // Quebra de linha no final
    return 0; // Finaliza o programa
}
```

- É um processo que: 
- Compara os elementos dois a dois,
- Troca de posição se estiverem fora de ordem,
- E repete até o vetor inteiro ficar ordenado.

## Explicando passo a passo:

1.  $5 > 2 \rightarrow$  troca,  $5 > 9 \rightarrow$  não troca,  $9 > 1 \rightarrow$  troca,  $9 > 6 \rightarrow$  troca
2.  $2 > 5 \rightarrow$  não troca,  $5 > 1 \rightarrow$  troca,  $5 > 6 \rightarrow$  não troca
3.  $2 > 1 \rightarrow$  troca,  $2 > 5 \rightarrow$  não troca
4.  $1 > 2 \rightarrow$  não troca

# Selection Sort



## Conceito:

- O **Selection Sort** funciona assim:
  - A cada passagem:
    - Procura o **menor elemento** no vetor restante.
    - **Traz o menor elemento para a frente.**
- Repete isso até ordenar todo o vetor.
- Fácil de entender também.
- Ainda não é muito rápido para grandes vetores.



## Quando usar?

- Vetores pequenos (poucos elementos)
- Ideal para **iniciantes** e ensino básico.

## Limitações:

- Lento para grandes vetores.
- Complexidade de tempo alta
- Pouco usado em aplicações profissionais
- Não é adaptativo

# Selection Sort

```
#include <stdio.h>    // Biblioteca padrão
#define TAM 5           // Tamanho do vetor
int main() {
    int numeros[TAM] = {29, 10, 14, 37, 13}; // Vetor desordenado
    int i, j, min, temp; // Variáveis auxiliares
    // Algoritmo Selection Sort
    for (i = 0; i < TAM - 1; i++) {           // Percorre cada posição do vetor
        min = i;                             // Assume que o menor elemento está na posição atual
        for (j = i + 1; j < TAM; j++) {         // Percorre o resto do vetor
            if (numeros[j] < numeros[min]) {   // Se encontrar um menor
                min = j;                      // Atualiza a posição do menor
            }
        }
        // Se o menor elemento não for o atual, faz a troca
        if (min != i) {
            temp = numeros[i];
            numeros[i] = numeros[min];
            numeros[min] = temp;
        }
    }
    // Imprime o vetor ordenado
    printf("Vetor ordenado: ");
    for (i = 0; i < TAM; i++) {
        printf("%d ", numeros[i]);
    }
    printf("\n");
    return 0;
}
```

Passada		
1 <sup>a</sup> Passada	10	Troca 29 por 10
2 <sup>a</sup> Passada	13	Troca 29 por 13
3 <sup>a</sup> Passada	14	Troca 29 por 14
4 <sup>a</sup> Passada	29	Sem troca

Menor encontrado
10
13
14
29

Troca
Troca 29 por 10
Troca 29 por 13
Troca 29 por 14
Sem troca

# QuickSort



## Conceito:

- O **QuickSort** é um dos **mais rápidos** algoritmos de ordenação para grandes quantidades de dados.
- Funciona assim:
  - Escolhe um **pivô** (um elemento qualquer).
  - **Divide** o vetor em dois:
    - Todos menores que o pivô ficam de um lado,
    - Todos maiores ficam do outro lado.
  - **Chama o mesmo processo** recursivamente para as duas partes.
- Muito eficiente para vetores grandes.  
 Usa **recursão**.

# O que é recursão?

- Recursão é quando uma função chama ela mesma para resolver um problema em partes menores.
- Em vez de resolver o problema inteiro de uma vez,  
 A função quebra o problema em pedaços menores,  
 E cada pedaço é resolvido chamando a mesma função novamente.

```
void contar(int n) {  
    if (n == 0) return;    // Condição de parada  
    printf("%d\n", n);    // Imprime o número  
    contar(n - 1);        // Chama a função novamente com n-1  
}
```

# QuickSort

```
#include <stdio.h> // Biblioteca padrão
#define TAM 6 // Tamanho do vetor
// Função para trocar dois elementos
void troca(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Função principal do QuickSort
void quicksort(int vetor[], int inicio, int fim) {
    if (inicio < fim) { // Condição de parada
        int pivo = vetor[inicio]; // Define o primeiro elemento como pivô
        int i = inicio + 1;
        int j = fim;
        while (i <= j) { // Enquanto os índices não se cruzarem
            if (vetor[i] <= pivo) { // Se o valor é menor ou igual ao pivô
                i++;
            } else if (vetor[j] > pivo) { // Se o valor é maior que o pivô
                j--;
            } else { // Se precisar trocar
                troca(&vetor[i], &vetor[j]); // Troca os elementos
                i++;
                j--;
            }
        }
        troca(&vetor[inicio], &vetor[j]); // Troca o pivô com o elemento na posição j
        quicksort(vetor, inicio, j - 1); // Ordena a parte esquerda
        quicksort(vetor, j + 1, fim); // Ordena a parte direita
    }
}
```

```
int main() {
    int numeros[TAM] = {20, 10, 40, 30, 60, 50}; // Vetor desordenado
    int i;

    quicksort(numeros, 0, TAM - 1); // Chama o Quicksort

    // Imprime o vetor ordenado
    printf("Vetor ordenado: ");
    for (i = 0; i < TAM; i++) {
        printf("%d ", numeros[i]);
    }
    printf("\n");

    return 0;
}
```

# Passo a passo da depuração

## 1 Primeira chamada:

- inicio = 0, fim = 5

- **Pivô = 20**

### Processo:

- i começa no 1 (10), j começa no 5 (50).

- Comparamos:

- $10 \leq 20 \rightarrow i++ \rightarrow$  agora  $i = 2$

- $40 > 20 \rightarrow j-- \rightarrow$  agora  $j = 4$

- $60 > 20 \rightarrow j-- \rightarrow$  agora  $j = 3$

- $30 > 20 \rightarrow j-- \rightarrow$  agora  $j = 2$

- $40 > 20 \rightarrow j-- \rightarrow$  agora  $j = 1$

- Agora  $i > j$  ( $2 > 1$ ), então trocamos o **pivô (20)** com o valor da posição  $j = 1$  (10).

### Resultado depois da troca:

[10, 20, 40, 30, 60, 50]

- Agora, 10 está antes de 20, e 20 está na posição certa (pos 1).

## 2 Segunda chamada:

- inicio = 2, fim = 5

- **Pivô = 40**

### Processo:

- $i = 3$  (30),  $j = 5$  (50)

- Comparamos:

- $30 \leq 40 \rightarrow i++ \rightarrow i = 4$

- $50 > 40 \rightarrow j-- \rightarrow j = 4$

- $60 > 40 \rightarrow j-- \rightarrow j = 3$

- Agora  $i > j$  ( $4 > 3$ ), então trocamos o pivô (40) com o elemento da posição  $j = 3$  (30).

### Resultado depois da troca:

[10, 20, 30, 40, 60, 50]

# Passo a passo da depuração

## ③ Terceira chamada:

- inicio = 4, fim = 5
- Pivô = 60

### Processo:

- i = 5 (50), j = 5 (50)
- Comparamos:
  - $50 \leq 60 \rightarrow i++ \rightarrow i = 6$
- Agora  $i > j$ , então trocamos o pivô (60) com o elemento na posição  $j = 5$  (50).

### Resultado depois da troca:

csharp

[10, 20, 30, 40, 50, 60]