Estruturas de Dados e Algoritmos

As **estruturas de dados** são essenciais para organizar dados em algoritmos, impactando diretamente o desempenho e a experiência do usuário. A escolha correta da estrutura de dados pode otimizar o processamento, enquanto uma escolha inadequada prejudica o desempenho.

- **Importância**: Escolher a estrutura certa melhora a eficiência e a experiência; uma estrutura inadequada leva a problemas de performance.
- **Objetivo do Estudo**: Compreender diferentes tipos de estruturas de dados para selecionar a mais adequada para cada problema.
- **Tópicos Abordados**:
    - **Ponteiros**: Introdução ao uso de ponteiros para manipulação eficiente de dados em memória.
    - **Estruturas Heterogêneas e Homogêneas**: Revisão dos conceitos e das diferenças entre estruturas que armazenam tipos de dados iguais (homogêneas) e diferentes (heterogêneas).
    - **Linguagem C**: Aplicação prática com ponteiros e `structs`, explorando structs aninhadas e vetores de structs para usos avançados.

Este conhecimento é essencial para desenvolver algoritmos eficientes e bem estruturados na linguagem C.

Ponteiros e Estrutura de Memória

Ponteiros

- Definição: Um ponteiro é uma variável especial que armazena um endereço de memória, podendo apontar para o endereço de outra variável.
- Utilidade: Permite acesso direto à memória, essencial para manipulação eficiente de dados e estruturas em programação.

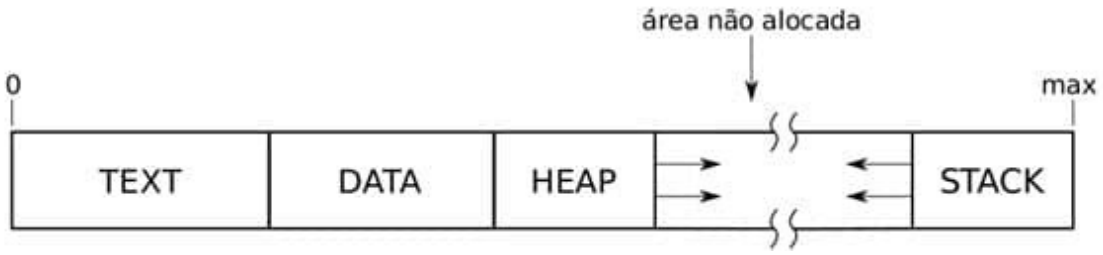
Memória do Computador

- Composta de palavras, cada uma identificada por um endereço único.
- Cada palavra armazena uma quantidade fixa de bytes.

Segmentos Lógicos da Memória

1. **TEXT**: Armazena o código do programa e constantes, com tamanho fixo durante a execução.
2. **DATA**: Contém variáveis globais e estáticas, também com tamanho fixo.
3. **STACK**: Pilha de execução que guarda parâmetros, endereços de retorno e variáveis locais, com tamanho variável.
4. **HEAP**: Bloco de memória para alocações dinâmicas, com tamanho variável conforme necessário.

Esses segmentos organizam a memória para que o programa gerencie dados e variáveis de forma estruturada e eficiente.



Tipos de Alocação de Memória em C

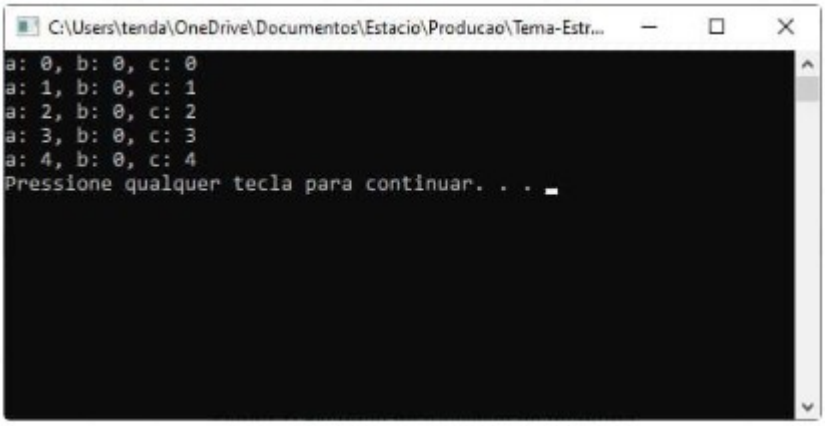
1. **Alocação Estática**: Variáveis globais e variáveis declaradas como `static` são armazenadas na área de dados (DATA) e mantêm seu valor durante toda a execução do programa, a menos que sejam explicitamente modificadas.
2. **Alocação Automática**: Variáveis locais, sem o modificador `static`, são alocadas na pilha (STACK) e perdem o valor após a execução da função em que foram declaradas.
3. **Alocação Dinâmica**: Usada para alocar memória em tempo de execução na área de heap (HEAP), com controle manual sobre a alocação e liberação de memória.

Exemplo de Alocação Estática

- No código de exemplo:
  - `a` (linha 3) é uma variável global estática.
  - `c` (linha 8) é uma variável local estática.
  - Ambas as variáveis preservam seus valores entre chamadas da função `incrementa`.
- **Execução**: A cada chamada da função `incrementa`, `a` e `c` mantêm seus valores incrementados, enquanto `b`, uma variável local automática, é redefinida a cada execução.

```c
#include < stdio.h >

    static int a = 0;  // variável global, alocação estática

    void incrementa(void)
    {
        int b = 0; // variável local, alocação automática
        static int c = 0; // variável local, alocação estática

        printf ("a: %d, b: %d, c: %d\n", a, b, c);
        a++;
        b++;
        c++;
    }

    int main(void)
    {
        int i;
        for (i = 0; i < 5; i++)
        incrementa();
        system("pause");
        return(0);
    }
```

A execução desse código gera a seguinte saída:



Alocação Automática

- **Definição**: Alocação automática ocorre para variáveis locais e parâmetros de função. As variáveis são alocadas na memória ao iniciar a função e liberadas ao final da execução dessa função.

- **Exemplo**: No Exemplo 1, a variável `b` (linha 7) é alocada automaticamente a cada chamada da função `incrementa` e descartada quando a função termina.
- **Comportamento**: Valores de variáveis automáticas não são preservados entre chamadas de função, ou seja, perdem seu valor ao fim da execução da função.

Alocação Dinâmica

- **Definição**: A alocação dinâmica permite ao programa solicitar áreas de memória conforme necessário e liberá-las quando não forem mais úteis. O programador controla essa memória manualmente, liberando-a com a função adequada.
- **Funções principais**:
  - `malloc(size_t size)`: Aloca uma quantidade especificada de bytes e retorna um ponteiro para a primeira posição. Retorna `NULL` caso a alocação falhe.
  - `free(void *ptr)`: Libera a área de memória apontada por `ptr`. Após liberar, recomenda-se definir o ponteiro para `NULL` para evitar erros.
  - `realloc(void *ptr, size_t newsize)`: Redimensiona uma área previamente alocada para um novo tamanho `newsize`, retornando o novo endereço.

```
ptr = malloc (1024);

...

free (ptr);

ptr = NULL; // não é obrigatório, mas aconselhável
```

- **Alocação por vetor**:
  - `calloc(size_t count, size_t eltSize)`: Aloca um bloco de memória para um vetor com `count` elementos, cada um de tamanho `eltSize`.

Tabela Resumo:

Função	Ação	Sintaxe
`malloc`	Aloca memória	`void *malloc(size_t size)`
`free`	Libera memória	`void free(void *ptr)`
`realloc`	Redimensiona memória	`void *realloc(void *ptr, size_t newsize)`
`calloc`	Aloca memória para vetor	`void *calloc(size_t count, size_t eltSize)`

Na **alocação dinâmica**, o programa pede explicitamente uma área na memória para armazenar dados. Isso é útil quando não se sabe o tamanho exato dos dados com antecedência. Uma vez que a memória não é gerenciada automaticamente, o programador precisa ter controle sobre a alocação e liberação, evitando ocupar memória desnecessária ou criar problemas de memória insuficiente.

Funções de Alocação Dinâmica:

1. **malloc (Memory Allocation)**:
   - A função `malloc` permite que o programa reserve um certo número de bytes na memória. Por exemplo, se `malloc(1024)` é usado, o programa tenta alocar 1024 bytes de memória.

- `malloc` retorna um ponteiro para o primeiro byte da área alocada. Se a alocação não for possível, ela retorna `NULL`, indicando que a memória não foi alocada.
- Exemplo: `ptr = malloc(1024);` reserva 1024 bytes e armazena o endereço inicial em `ptr`.

2. **free (Liberar Memória)**:
   - A função `free` libera a área de memória anteriormente alocada com `malloc`. Isso ajuda a reutilizar a memória e a evitar vazamentos (quando a memória é ocupada, mas nunca liberada).
   - Exemplo: `free(ptr);` libera o espaço apontado por `ptr`.
   - Como prática recomendada, o ponteiro `ptr` é definido como `NULL` após `free`, pois ele continua apontando para a área de memória liberada, o que pode causar erros se for reutilizado acidentalmente.
3. **realloc (Realocar Memória)**:
   - A função `realloc` redimensiona uma área de memória já alocada, alterando seu tamanho conforme necessário.
   - Se a nova área alocada precisar de mais espaço e o local original não tiver essa capacidade, `realloc` pode mover os dados para um novo local, retornando o novo endereço.
   - Exemplo: `ptr = realloc(ptr, 2048);` redimensiona `ptr` para 2048 bytes.
4. **calloc (Alocar para Vetor)**:
   - A função `calloc` aloca memória para uma coleção de elementos, ou seja, um vetor, garantindo que cada elemento tenha um tamanho específico e que a memória seja inicializada com zero.
   - Exemplo: `ptr = calloc(10, sizeof(int));` aloca espaço para um vetor com 10 inteiros, todos com valor zero inicial.

c Copiar

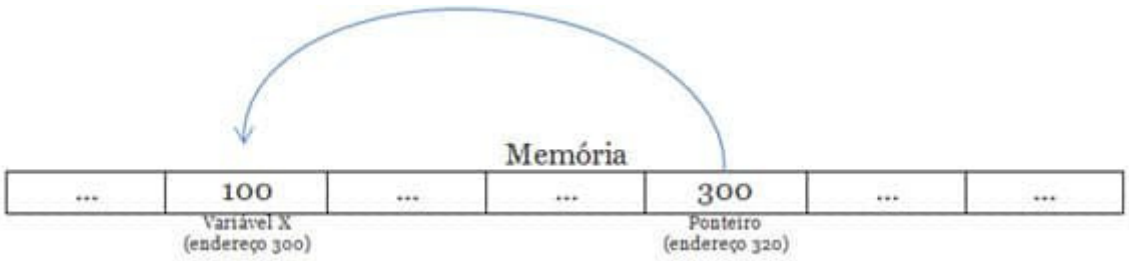
```c
#include <stdlib.h>

void main() {
    int *ptr;
    ptr = malloc(1024);       // Aloca 1024 bytes
    if (ptr != NULL) {
        // Usa a memória
    }
    free(ptr);                // Libera a memória
    ptr = NULL;               // Evita erros de uso após liberação

    ptr = calloc(10, sizeof(int)); // Aloca um vetor com 10 inteiros
    ptr = realloc(ptr, 20 * sizeof(int)); // Aumenta para um vetor com 20 inteiros
}
```

Ponteiros em C

Em C, cada variável possui um **nome**, **tipo**, **valor** e **endereço de memória**. Por exemplo, na declaração `int x = 5;`, "x" é o nome da variável, seu tipo é `int`, o valor é 5, e ela está armazenada em um endereço específico na memória, como o endereço 10. Variáveis com tipos diferentes ocupam tamanhos distintos de memória; `int` geralmente ocupa 2 bytes, enquanto `char` ocupa 1 byte. O endereço de uma variável indica o primeiro byte que ela ocupa na memória, como no caso de `x` armazenado no endereço 10 e `c` no endereço 13.

Ponteiros e Acesso Indireto

Na linguagem C, cada dado em memória possui um endereço único. Um **ponteiro** é uma variável especial que armazena esse endereço de outra variável, em vez de armazenar o valor diretamente. Por exemplo, se a variável `x` armazena o valor `100` no endereço `300`, um ponteiro `p` pode armazenar o endereço `300` e, assim, apontar para `x`. Isso permite o **acesso indireto**, que é uma das vantagens dos ponteiros: é possível ler ou modificar o valor de `x` usando `p`, sem precisar referenciar `x` diretamente.



Declaração e Operadores de Ponteiros em C

1. **Declaração de Ponteiros**:
   - Os ponteiros são declarados usando o símbolo `*`, que é colocado entre o tipo de dado e o nome da variável.
   - **Forma Geral**: `Tipo_da_variável * Nome_da_Variável;`
   - **Exemplos**:
     - `int *p;` // Ponteiro para um inteiro
     - `float *q;` // Ponteiro para um float
     - `char *r;` // Ponteiro para um caractere
2. **Operadores Relacionados a Ponteiros**:
   - **Operadores Unários de Ponteiros**

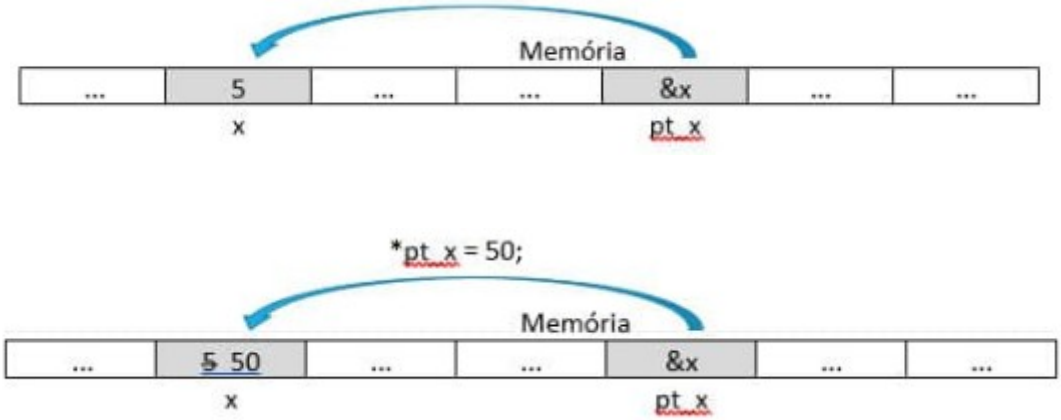
- O operador unário “&” obtém o endereço de memória de uma variável.
- O operador unário “*” realiza a *deferência*, acessando o valor armazenado no endereço referenciado pelo ponteiro.

Exemplo:

- No código `int x = 5; int *pt_x;`, o ponteiro `pt_x` recebe o endereço de `x` com `pt_x = &x;`. Assim, `pt_x` armazena o endereço de `x`, permitindo manipular `x` indiretamente.
- Para acessar ou modificar o valor de `x` indiretamente, utiliza-se `*pt_x`.

Exemplo de Modificação:

- A instrução `*pt_x = 50;` altera o valor de `x` para 50 via ponteiro, sem referenciar diretamente `x`. Isso caracteriza `pt_x` como um ponteiro variável, pois permite armazenar qualquer endereço e manipular o valor indiretamente.



Operadores de Indireção Múltipla em Ponteiros

- Um ponteiro pode armazenar o endereço de outro ponteiro, criando *indireção múltipla*.
- A indireção de um ponteiro é definida pelo número de operadores `*` usados. Exemplo: `int **pt2;` define um ponteiro para ponteiro do tipo inteiro.

Exemplo:

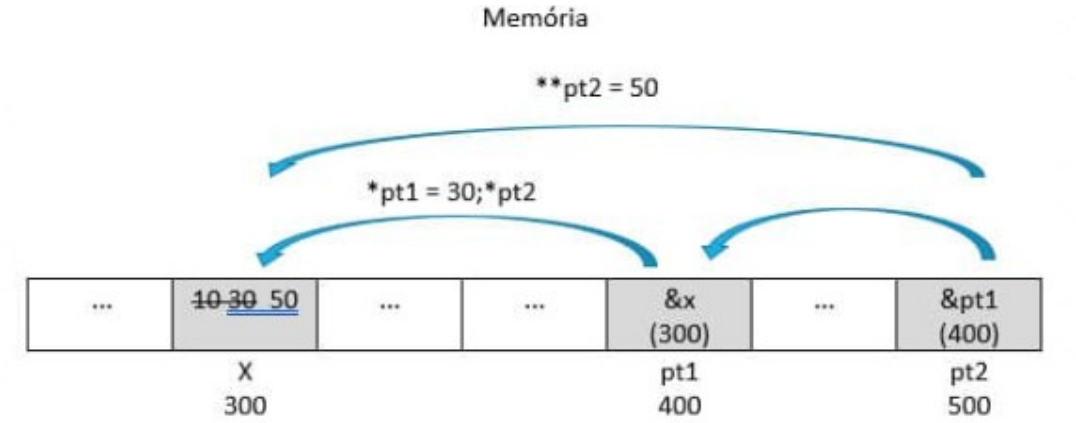
- Declarações e valores iniciais:

```
c

int **pt2; // ponteiro para ponteiro do tipo inteiro
int *pt1;  // ponteiro para inteiro
int x = 10;
pt2 = &pt1; // pt2 recebe o endereço de pt1
pt1 = &x;   // pt1 recebe o endereço de x
```

- Acessos e modificações indiretas:
  - `*pt1 = 30;` altera o valor de `x` para 30.
  - `**pt2 = 50;` altera `x` para 50, acessando-o indiretamente via `pt2`.

Definições dos Elementos:

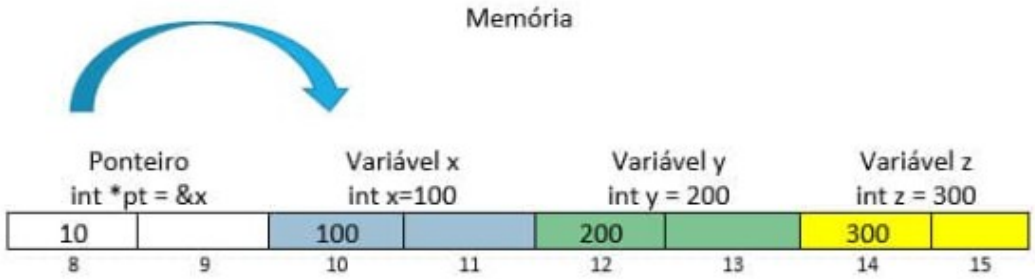
- `&pt2` e `&pt1`: Endereços dos ponteiros `pt2` e `pt1`.
- `pt2`: Contém o endereço de `pt1`.
- `pt1`: Contém o endereço de `x`.
- `*pt2`: Acessa `pt1`.
- `*pt1`: Acessa `x`.
- `**pt2`: Acessa o valor de `x` indiretamente através de `pt2`.

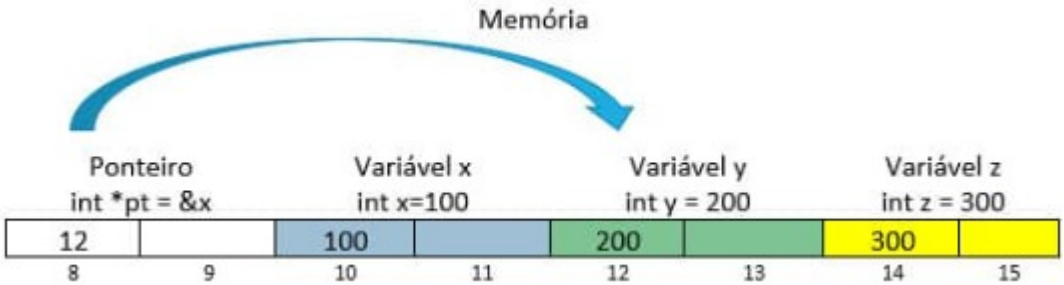


Aritmética de Ponteiros

- Em ponteiros, só são permitidas operações de *adição* e *subtração*, válidas quando os ponteiros apontam para o mesmo tipo de dado.
- *Incremento (pt++):* Faz o ponteiro apontar para o próximo elemento do mesmo tipo, aumentando o endereço conforme o tamanho do tipo de dado. Exemplo: se `pt` aponta para `x` no endereço 10, `pt++` fará apontar para `y` no endereço 12, caso `int` ocupe 4 bytes.
- *Decremento (pt--):* Faz o ponteiro apontar para o elemento anterior.
- Qualquer operação com ponteiros considera o *tamanho do tipo base*. Exemplo: `pt = pt + 4` desloca o ponteiro 16 bytes (4 x 4 bytes) em um ponteiro para `int` (4 bytes).

Essas operações permitem navegar entre elementos de dados sequenciais.





Utilização dos Ponteiros

Ponteiros permitem acessar variáveis em diferentes partes do programa e são essenciais em diversas situações, como:

- Alocação dinâmica de memória, onde é possível reservar e liberar memória conforme a necessidade durante a execução.
- Manipulação de arrays, possibilitando navegar pelos elementos através de aritmética de ponteiros.
- Retorno de múltiplos valores em funções, onde ponteiros podem passar o endereço de variáveis que serão modificadas pela função.
- Estruturas de dados como listas, pilhas, árvores e grafos, pois permitem criar ligações dinâmicas entre elementos.

Exemplo 4: Implementação de Ponteiro em C

- O exemplo define duas variáveis: `v_num` como um `int`, e `ptr` como um ponteiro para `int` (`int *ptr`).
- Na linha 13, o endereço de `v_num` é atribuído a `ptr` usando `&v_num`. O operador `&` obtém o endereço de memória de `v_num`, que o ponteiro `ptr` armazena.
- A declaração `int *ptr` indica que `ptr` é um ponteiro para uma variável do tipo `int`, pois o tipo do ponteiro deve ser compatível com o tipo da variável que ele aponta.

```
1  #include < stdio.h >
2  #include < conio.h >
3  int main(void)
4  {
5  //v_num é a variável que
6  //será apontada pelo ponteiro
7  int v_num = 10;
8
9  //declaração de variável ponteiro
10  int *ptr;
11
12  //atribuindo o endereço da variável v_num ao ponteiro
13  ptr = &v_num;
14
15  printf("Utilizando ponteiros\n\n");
16  printf("Conteúdo da variável v_num: %d\n", v_num);
17  printf("Endereço da variável v_num: %x \n", &v_num);
18  printf("Conteúdo da variável ponteiro ptr: %x", ptr);
19
20  getch();
21  return(0);
```

Impressão dos Valores

- Nas linhas 15 a 18, o código imprime:
  - O conteúdo de `v_num`, que é o valor 10.
  - O endereço de memória de `v_num`, que `&v_num` obtém.
  - O conteúdo de `ptr`, que é o endereço de `v_num`, mostrado em hexadecimal com `%x`.
- O uso de `%x` é necessário para exibir valores de endereço, pois são em formato hexadecimal.

Resumo do Funcionamento do Código Esse exemplo ilustra como `ptr` age como um "ponteiro variável" que armazena o endereço de `v_num`. A manipulação de `ptr` permite acessar e modificar `v_num` indiretamente, uma técnica usada para manipulação eficiente de dados na memória.

Estruturas de dados são métodos para organizar e armazenar dados no computador de forma que permitam acesso e processamento eficientes. Elas representam coleções de informações que podem ser manipuladas por programas, definindo como os dados estão organizados, os métodos de acesso disponíveis e as opções de processamento. Essa organização melhora a eficiência dos algoritmos que lidam com esses dados, atendendo a diferentes requisitos de processamento.

Um algoritmo é uma sequência finita e organizada de passos que executa uma tarefa específica. Ele funciona como um conjunto de instruções que permite que dispositivos, como computadores,

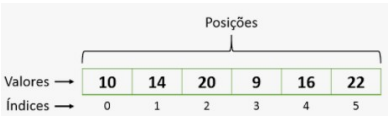
realizem ações. Um programa, por sua vez, é um código composto por algoritmos que definem o que deve ser feito.

Algoritmos manipulam dados e, quando esses dados estão organizados de forma lógica, formam uma estrutura de dados. Assim, ao criar um algoritmo para resolver um problema, o programador também define uma estrutura de dados para ser manipulada pelo algoritmo.

As estruturas de dados organizam e armazenam dados para uso eficiente. Dividem-se em **dado** (elemento com valor para resolver problemas computacionais) e **estrutura** (organização que carrega as informações). Os tipos básicos de dados são:

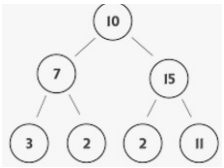
- **Inteiro**: números inteiros, sem decimais.
- **Real**: números com decimais, também conhecidos como ponto flutuante.
- **Lógico**: valores booleanos (verdadeiro/falso).
- **Texto**: sequência de caracteres, subdividida em string (sequência) e caractere único; tipo real subdividido em float (precisão simples) e double (precisão dupla).

Principais Estruturas de Dados:



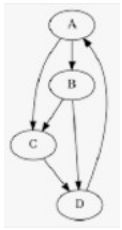
- **Array (Vetor)**: armazena dados homogêneos em uma sequência indexada; arrays unidimensionais têm uma linha; bidimensionais formam uma matriz com linhas e colunas.

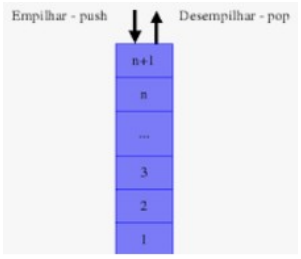
- **Fila (Queue)**: organiza elementos em uma sequência FIFO (primeiro a entrar, primeiro a sair). Inserções ocorrem no final, remoções no início.



- **Árvore**: representa relações hierárquicas, como organogramas, onde cada elemento (nó) se relaciona de forma única com outros.
- **Lista Encadeada**: armazena dados em sequência não contígua na memória, sendo flexível e eficiente para operações dinâmicas.

- **Pilha (Stack)**: estrutura LIFO (último a entrar, primeiro a sair), onde o último elemento adicionado é o primeiro a ser removido. Necessita remoção de elementos acima para acessar o último elemento.





- **Grafo**: estrutura complexa que conecta elementos de modo relacional, ideal para representar redes, como trajetos entre cidades, onde cada elemento (nó) conecta-se a outros.

Essas estruturas são essenciais para implementar e otimizar algoritmos e resolver problemas complexos em programação.

As estruturas de dados podem ser **homogêneas** ou **heterogêneas**:

- **Estruturas Homogêneas**: Formadas por elementos do mesmo tipo, permitem agrupar vários valores em uma única variável. Exemplos incluem vetores e matrizes.
  - **Vetor**: Estrutura linear e estática, com elementos armazenados em posições contíguas de memória. Cada posição é acessada por um índice. Exemplo: um vetor de notas com 9 posições.

- **Matriz**: Estrutura multidimensional que requer múltiplos índices para acessar cada elemento, sendo uma extensão dos vetores em múltiplas dimensões.

Vetores usam um índice único para acesso a elementos, enquanto matrizes exigem um índice por dimensão.

Estruturas de dados **heterogêneas** são compostas por dados de tipos diferentes, como registros. Um **registro** pode conter diversos campos, cada um com um tipo distinto, como em uma agenda telefônica (nome, telefone, endereço).

Registro Funcionário	
Matrícula	Tipo Inteiro
Nome	Tipo Cadeia de Caracteres
Dt. Nascimento	Tipo Data
Cargo	Tipo Cadeia de Caracteres
Salário	Tipo Real

Exemplo: um cadastro de funcionário com campos como matrícula, nome, data de nascimento, cargo e salário, cada um de um tipo diferente (inteiro, cadeia de caracteres, data, real).

A definição de campos em um registro é similar à declaração de variáveis, permitindo declarar múltiplos campos do mesmo tipo em uma única linha. A estrutura de declaração de um registro segue o formato:

```ruby
<nome_variavel>:registro
Início
    <campo1>:<tipo1>
    <campo2>:<tipo2>
    ...
Fimregistro
```

Exemplo de declaração de registro para ficha de cadastro de funcionários:

```yaml
var Ficha_Funcionario: registro
inicio
    matricula: inteiro
    nome: vetor[1..50] de caractere
    Dt_Nascimento: vetor[1..9] de caractere
    Cargo: vetor[1..30] de caractere
    Salario: real
fim
```

A variável `Ficha_Funcionario` agrupa as informações de um funcionário, tratando os dados como uma única variável. Para acessar campos de um registro, utiliza-se `varReg.campo`, por exemplo: `Ficha_Funcionario.matricula`.

Diferente de vetores, registros permitem atribuições diretas entre variáveis, copiando o conteúdo de todos os campos, inclusive vetores.

Registros são úteis para tabelas heterogêneas ou arquivos binários. Para armazenar múltiplos registros, usa-se vetores de registro. Exemplo:

```yaml
var funcionario: vetor[1..50] de registro
Inicio
    matricula: inteiro
    nome: caractere
    dtNascimento: caractere
    cargo: caractere
    salario: real
fimregistro
```

Aqui, um vetor de registros representa uma tabela com 50 linhas (funcionários) e 5 colunas (dados: matrícula, nome, data de nascimento, cargo, salário).

Na linguagem C, os tipos de dados podem ser básicos (int, char, float, double, void) ou compostos homogêneos (arrays). Quando esses tipos não são suficientes, é possível criar novos tipos, como os registros, usando a estrutura `struct`.

A `struct` é uma coleção de variáveis relacionadas com um nome comum e pode conter variáveis de tipos diferentes, ao contrário dos arrays, que são homogêneos. As `structs` são úteis para definir registros e, junto com ponteiros, ajudam a criar estruturas de dados complexas, como pilhas, listas ligadas, filas e árvores.

A definição de uma `struct` segue a sintaxe:

```c
struct identificador {
   tipo variável;
   tipo variável;
   tipo variável;
};
```

Exemplos:

```c
struct mystruct {
   char a;
   int b;
   float c;
};
```

```c
struct endereco {
   char rua[50];
   char numero[5];
   char CEP[8];
   char bairro[30];
};
```

Cada variável dentro de uma `struct` pode ter um tipo diferente.

A declaração de uma estrutura de dados (`struct`) em C é feita com a seguinte sintaxe:

```c
struct nome_da_estrutura {
    tipo_campo1 nome_campo1;
    tipo_campo2 nome_campo2;
    ...
} variáveis_que_armazenam_a_estrutura;
```

`struct`: palavra-chave para definir a estrutura.

- **Tag da estrutura**: nome que identifica a estrutura e é usado para declarar variáveis desse tipo.
- **Campos ou membros**: variáveis dentro da estrutura, podendo ser de tipos básicos ou compostos (como arrays ou outras structs), mas não podem se referir a si mesmas.

A estrutura define o formato dos dados, mas não declara as variáveis diretamente. Após a definição, as variáveis podem ser declaradas separadamente.

Existem três formas de declarar uma `struct`:

1. Declarando a variável fora da estrutura:

```c
struct endereco x;
```

2. Declarando variáveis ao definir a estrutura:

```c
struct livro {
    char nome[30];
    char autor[50];
    int paginas;
    float preco;
} livro1, livro2, livro3;
```

3. Definindo uma variável sem nome para a estrutura:

```c
```

```
struct {
    char nome[30];
    char autor[50];
    int paginas;
    float preco;
} livro;
```

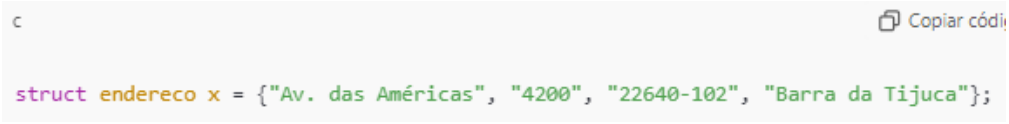
Na última forma, o nome da `struct` é opcional, mas sempre forneça um nome como prática recomendada.

Operações comuns com `structs` em C incluem:

- Atribuição entre variáveis do mesmo tipo.
- Leitura do endereço com o operador &.
- Acesso aos membros de uma variável de estrutura.
- Uso do operador `sizeof` para determinar o tamanho de uma estrutura.

Em C, uma `struct` pode ser inicializada de maneiras semelhantes aos vetores e matrizes:

1. Lista de inicialização: Os membros são inicializados na ordem declarada na `struct`. Se houver menos inicializadores que membros, os restantes são inicializados com zero ou `NULL` (para ponteiros). Exemplo:



2. Atribuição de uma estrutura já inicializada: Pode-se atribuir a uma variável de `struct` o valor de outra já inicializada.

3. Atribuição aos membros individualmente: Atribuindo valores aos campos da `struct` usando a notação `nome_da_struct.membro`. Exemplo:

Para acessar membros de uma `struct` em C, usam-se dois operadores:

1. **Operador ponto (.)**: Usado para acessar membros diretamente de uma variável do tipo `struct`. Exemplo:

   c

   ```
   printf("%s", x.rua);
   ```

2. **Operador seta (->)**: Usado quando a `struct` é referenciada por um ponteiro. Exemplo:

   c

   ```
   printf("%s", xptr->rua);
   ```

O operador ponto é utilizado com variáveis de `struct`, enquanto o operador seta é utilizado com ponteiros que apontam para `struct`.

As `structs` em C permitem manipulação direta de seus campos, seja para atribuição, impressão ou entrada de dados.

Atribuição de valores

Para atribuir valores a campos de uma `struct`, basta referenciá-los diretamente:

```
aluno_especial.codigo = 10;
strcpy(aluno_especial.nome, "Manoel");
aluno_especial.nota = 10.0;
```

Obs.: Para atribuir uma string, use `strcpy`.

Strcpy (CPY = copiar; STR = string). A função copiará o que está dentro das aspas duplas para o campo STRING da estrutura.

Impressão de campos

Para imprimir campos, utilize `printf`. Exemplo:

```
printf("\n%d", aluno_especial.codigo);
printf("\n%s", aluno_especial.nome);
printf("\n%.2f", aluno_especial.nota);
```

Para imprimir todos os membros de uma `struct` de uma vez, recomenda-se uma função:

```
void imprimir(Aluno aluno_regular) {
    printf("\n%d", aluno_regular.codigo);
    printf("\n%s", aluno_regular.nome);
    printf("\n%.2f", aluno_regular.nota);
}
```

Especificador	Significado	Exemplo
`%f`	Exibe ponto flutuante com 6 casas decimais padrão	`123.456000`
`%.nf`	Define `n` casas decimais	`%.2f` → `123.46`
`%e` ou `%E`	Exibe ponto flutuante em notação científica	`1.23e+02`
`%g`	Escolhe automaticamente `%f` ou `%e`	`123.46`

Chamada da função: `imprimir(aluno_especial);`

%d é um **especificador de formato** usado em funções como **`printf`** ou **`scanf`** em C para manipular números inteiros do tipo **`int`**.

- **%: Indica que estamos especificando um formato para saída ou entrada.**
- **`d`**: Refere-se a "**decimal**" (base 10), ou seja, números inteiros positivos ou negativos.

1. Use `%ld` para **`long int`**.

1. Use `%hd` para **`short int`**.
1. Use `%u` para exibir inteiros **sem sinal**.

Entrada de dados

Use `scanf` para obter valores do teclado:

```
printf("Digite o código: ");
scanf("%d%*c", &aluno_especial.codigo);
printf("Digite o nome: ");
scanf("%s%*c", aluno_especial.nome);
printf("Digite a nota: ");
scanf("%f%*c", &aluno_especial.nota);
```

Ou crie uma função `cadastrar` para entradas:

```
void cadastrar(Aluno aluno_especial) {
    printf("Digite o código: ");
    scanf("%d%*c", &aluno_especial.codigo);
    printf("Digite o nome: ");
    scanf("%s%*c", aluno_especial.nome);
    printf("Digite a nota: ");
    scanf("%f%*c", &aluno_especial.nota);
}
```

O formato **%d%*c** é usado em funções como scanf para lidar com uma entrada onde:

1. **%d**: Lê um número inteiro da entrada e o armazena em uma variável do tipo `int`.
2. **%*c**: Ignora um caractere imediatamente após o número lido. O caractere não é armazenado nem usado.

```
%*c: "Leia um caractere, mas não o armazene em nenhuma variável."
```

Ao usar **scanf**, o caractere **\n** (ou qualquer outro caractere após o número) pode ficar na entrada e interferir em leituras subsequentes. Por exemplo, se você deseja ler um caractere depois de um número, sem o %*c, o programa pode interpretar o **\n** como a entrada desse caractere.

Nas funções `imprimir` e `cadastrar`, a `struct` é passada por valor, sendo necessária ou não, conforme a aplicação.

```c
/* Cria uma estrutura para armazenar dados de um aluno */
#include <stdio.h>
#include <stdlib.h>

struct aluno {
    int v_nmat;       // Número da matrícula
    float v_nota[3];  // Array para armazenar três notas
    float v_media;    // Média das notas
};

int main() {
    struct aluno Felipe;  // Declara uma variável do tipo struct aluno

    // Atribui valores aos campos da estrutura
    Felipe.v_nmat = 120;             // Atribui 120 ao campo v_nmat (matrícula)
    Felipe.v_nota[0] = 8.5;          // Atribui 8.5 à primeira nota
    Felipe.v_nota[1] = 7.2;          // Atribui 7.2 à segunda nota
    Felipe.v_nota[2] = 5.4;          // Atribui 5.4 à terceira nota

    // Calcula a média das notas
    Felipe.v_media = (Felipe.v_nota[0] + Felipe.v_nota[1] + Felipe.v_nota[2]) / 3.0;

    // Imprime a matrícula e a média do aluno
    printf("Matricula: %d\n", Felipe.v_nmat);
    printf("Media: %.2f\n", Felipe.v_media);

    system("pause");  // Pausa o sistema (relevante em alguns sistemas operacionais para visualizar a saída)
    return 0;
```

Explicação do Código

1. **Declaração da Struct**:

    - A estrutura `aluno` é declarada com três campos: `v_nmat` (int) para matrícula, `v_nota` (float[3]) como array para as notas e `v_media` (float) para a média das notas.
2. **Criação da Variável `Felipe`**:
    - A linha `struct aluno Felipe;` cria uma variável `Felipe` que utiliza a estrutura `aluno` como seu tipo.
3. **Atribuição de Valores**:
    - `Felipe.v_nmat = 120;` define o número de matrícula.
    - `Felipe.v_nota[0]`, `Felipe.v_nota[1]` e `Felipe.v_nota[2]` recebem as notas `8.5`, `7.2` e `5.4`, respectivamente.
4. **Cálculo da Média**:
    - A linha `Felipe.v_media = (Felipe.v_nota[0] + Felipe.v_nota[1] + Felipe.v_nota[2]) / 3.0;` calcula a média das notas e armazena no campo `v_media`.
5. **Impressão dos Resultados**:
    - `printf("Matricula: %d\n", Felipe.v_nmat);` imprime o número de matrícula.
    - `printf("Media: %.2f\n", Felipe.v_media);` imprime a média das notas com duas casas decimais.

Funcionamento do Programa

Ao executar o programa, ele exibe o número de matrícula e a média das notas do aluno `Felipe`. Este exemplo demonstra a criação de uma `struct`, atribuição de valores aos seus campos, cálculo de média e exibição dos dados usando a linguagem C.

```c
/* Ficha de Aluno */
#include <stdio.h>
#include <conio.h>

int main(void)
{
    /* Criando a struct */
    struct ficha_de_aluno
    {
        char nome[50];
        char disciplina[30];
        float nota_prova1;
        float nota_prova2;
    };

    /* Criando a variável aluno que será do tipo struct ficha_de_aluno */
    struct ficha_de_aluno aluno;

    printf("\n---------- Cadastro de aluno ----------\n\n\n");
    printf("Nome do aluno ......: ");
    fflush(stdin);
    fgets(aluno.nome, 40, stdin);
    printf("Disciplina ......: ");
    fflush(stdin);
    fgets(aluno.disciplina, 40, stdin);
    printf("Informe a 1a. nota ..: ");
    scanf("%f", &aluno.nota_prova1);
    printf("Informe a 2a. nota ..: ");
    scanf("%f", &aluno.nota_prova2);
```

```c
    printf("\n\n --------- Lendo os dados da struct ---------\n\n");
    printf("Nome ............: %s", aluno.nome);
    printf("Disciplina .....: %s", aluno.disciplina);
    printf("Nota da Prova 1 ...: %.2f\n", aluno.nota_prova1);
    printf("Nota da Prova 2 ...: %.2f\n", aluno.nota_prova2);

    getch();
    return 0;
}
```

Explicação do código:

1. **Declaração da Struct (`ficha_de_aluno`)**: Definida com os campos `nome`, `disciplina`, `nota_prova1` e `nota_prova2` para armazenar os dados de um aluno.
2. **Entrada de Dados**: `fgets()` lê o nome e a disciplina, e `scanf()` captura as notas.
3. **Saída de Dados**: Os dados da struct são impressos no formato adequado, mostrando o nome, disciplina e as duas notas.

```
---------- Cadastro de aluno ----------

Nome ............: Alana
Disciplina ......: Inteligência Artificial
Nota da Prova 1 .: 8
Nota da Prova 2 .: 9


 --------- Lendo os dados da struct ---------

Nome ............: Alana
Disciplina ......: Inteligência Artificial
Nota da Prova 1 .: 8.00
Nota da Prova 2 .: 9.00
```

Uma struct aninhada é uma estrutura de dados dentro de outra, permitindo que uma struct contenha outras structs como campos. Esse recurso facilita a criação de estruturas complexas que organizam dados de forma hierárquica. O padrão ANSI C define um limite de aninhamento de até 15 níveis, embora a maioria dos compiladores suporte níveis adicionais.

A declaração de structs define um tipo struct, nomeado com uma tag para referenciá-lo. Cada unidade de dados na struct é um membro, e os membros podem ser de qualquer tipo, incluindo outras structs, permitindo a criação de estruturas aninhadas.

Há duas formas de declarar structs aninhadas:

1. **Struct dentro de outra**: Coloca-se uma struct diretamente dentro de outra. A sintaxe é:

```
typedef struct {
    tipo membro_1;
    tipo membro_2;
    ...
    tipo membro_n;
    struct {
        tipo membro_interno_1;
        tipo membro_interno_2;
        ...
    };
} Nome_estrutura;
Nome_estrutura NE;
```

   Para acessar membros:

```
NE.membro_interno_1 = 0;  // Atribuição
&NE.membro_interno_1;      // Leitura do teclado
NE.membro_interno_1;       // Impressão
```

2. **Structs separadas, uma referenciada na outra**: Define-se uma struct primeiro e, em seguida, cria-se outra que contém uma variável da primeira. A sintaxe é:

```
typedef struct {
    tipo membro_1;
    tipo membro_2;
    ...
} nome_estrutura_1;

typedef struct {
    tipo membro_1;
    tipo membro_2;
    nome_estrutura_1 NE1;
    ...
} nome_estrutura_2;

nome_estrutura_2 NE2;
```

   Para acessar membros:

```
NE2.membro1;
NE2.NE1.membro1;
```

Essas estruturas aninhadas são úteis para organizar dados complexos em níveis.

Declaração de Structs Aninhadas:

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

struct departamento {
    int cod;
    char descricao[30];
};

struct cargo {
    int cod;
    char descricao[30];
};

struct funcionario {
    int cod;
    char nome[30];
    float salario;
    struct departamento depto;  // Membro do tipo struct departamento
    struct cargo cargo;          // Membro do tipo struct cargo
};

struct funcionario Funcionario;

int main(void) {
    // Corpo do programa
}
```

Explicação do código:

- **Estrutura `departamento`** (linhas 4-7) e **estrutura `cargo`** (linhas 9-12) são definidas com membros `cod` (int) e `descricao` (char[]).
- **Estrutura `funcionario`** (linhas 14-20) contém:
    - Membros simples: `cod` (int), `nome` (char[]), `salario` (float).
    - Membros aninhados: `depto` (do tipo `struct departamento`) e `cargo` (do tipo `struct cargo`).

Este exemplo ilustra como **structs aninhadas** permitem incluir uma estrutura dentro de outra. As structs `departamento` e `cargo` são aninhadas dentro de `funcionario` através dos campos `depto` e `cargo`.

Exemplo 8 - Declaração de Struct Aninhada (Forma Literal):

c

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

struct cargo {
    int cod;
    char descricao[30];
};

struct funcionario {
    int cod;
```

```
    char nome[30];
    float salario;
    struct departamento {   // Struct aninhada dentro de funcionario
        int cod;
        char descricao[30];
    };
    struct cargo cargo;     // Struct aninhada
};

struct funcionario Funcionario;

int main(void) {
    // Corpo do programa
}
```

Explicação:

- A estrutura `cargo` é declarada com dois membros: `cod` (int) e `descricao` (char[30]).
- A estrutura `funcionario` possui membros simples (como `cod`, `nome`, `salario`) e inclui duas structs aninhadas:
    - Uma estrutura aninhada chamada `departamento` (linhas 13-16).
    - Outra estrutura aninhada chamada `cargo` (linha 18).
- A estrutura `funcionario` é então instanciada na variável `Funcionario`.

Exemplo 8 - Adaptado com `typedef`:

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

typedef struct departamento {
    int cod;
    char descricao[30];
} Departamento;

typedef struct cargo {
    int cod;
    char descricao[30];
} Cargo;

typedef struct funcionario {
    int cod;
    char nome[30];
    float salario;
    Departamento depto;  // Referência a tipo typedef
    Cargo cargo;          // Referência a tipo typedef
} Funcionario;

int main(void) {
    // Corpo do programa
}
```

Explicação:

- A estrutura `departamento` é agora declarada com `typedef`, permitindo usar `Departamento` como tipo direto.

- A estrutura `cargo` também é declarada com `typedef`, permitindo usar `Cargo` como tipo.
- A estrutura `funcionario` agora usa os tipos `Departamento` e `Cargo` diretamente para os campos `depto` e `cargo`.

Esse exemplo mostra duas formas de declaração de structs aninhadas: uma em que a struct é declarada dentro de outra e outra utilizando `typedef` para simplificar o uso dos tipos.

O uso de `typedef` no código permite criar novos nomes para tipos de dados, simplificando a codificação sem afetar o desempenho. No exemplo, `typedef` é utilizado para definir os tipos `Departamento`, `Cargo` e `Funcionario`. Com isso, os campos `depto` e `cargo` da estrutura `Funcionario` passam a ser do tipo `Departamento` e `Cargo`, respectivamente, em vez de serem diretamente definidos como `struct`.

- `typedef` facilita a declaração de tipos definidos, permitindo usá-los como tipos de membros dentro de uma estrutura.
- Não há limite para o número de níveis em declarações aninhadas, mas adicionar muitos níveis pode tornar o código difícil de entender e manter.

Para manipular structs aninhadas, acessa-se seus membros usando o operador `.` repetidamente. No exemplo, temos duas estruturas: `endereco` e `student`, sendo que `student` contém membros do tipo `endereco`.

c

```
#define LEN 50

struct endereco {
    char rua[LEN];
    char cidade_estado_cep[LEN];
};

struct student {
    char id[10];
    int idade;
    struct endereco casa;
    struct endereco escola;
};

struct student pessoa;
```

Na estrutura `endereco`, temos os campos `rua` e `cidade_estado_cep`. Na estrutura `student`, temos os campos `id`, `idade`, e dois membros do tipo `endereco`: `casa` e `escola`.

Para acessar os membros da variável `pessoa` (do tipo `struct student`), utilizamos a seguinte sintaxe:

- `pessoa.id`
- `pessoa.casa.rua`
- `pessoa.casa.cidade_estado_cep`
- `pessoa.escola.rua`
- `pessoa.escola.cidade_estado_cep`

O operador `.` é usado para acessar os membros dentro das structs aninhadas.

No **Exemplo 9**, temos a declaração de uma estrutura `Aluno` que contém uma estrutura interna para armazenar a data de nascimento. A estrutura interna não tem nome, sendo diretamente definida dentro de `Aluno`.

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

typedef struct {
    int codigo;
    char nome[200];
    struct {
        int dia;
        int mes;
        int ano;
    };
} Aluno;

Aluno aluno;

int main(void) {
    setlocale(LC_ALL,"portuguese");
    aluno.codigo = 0;
    strcpy(aluno.nome, "NULL");
    aluno.dia = 0;
    aluno.mes = 0;
    aluno.ano = 0;
    printf("O código do aluno é: %d\n", aluno.codigo);
    printf("O nome do aluno é: %s\n", aluno.nome);
    printf("A data de nascimento do aluno é: %d / %d / %d\n", aluno.dia,
aluno.mes, aluno.ano);
    // Leitura de dados...
    return 0;
}
```

Aqui, a estrutura interna para a data de nascimento não tem nome, mas é acessada diretamente dentro de `Aluno`, como `aluno.dia`, `aluno.mes`, `aluno.ano`.

Já no **Exemplo 10**, a estrutura `Data` é declarada separadamente antes de ser utilizada em `Aluno`. Isso exige que a estrutura `Data` seja definida primeiro para ser referenciada corretamente.

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

typedef struct {
    int dia;
    int mes;
    int ano;
} Data;

typedef struct {
    int codigo;
    char nome[200];
    Data datNasc;
} Aluno;
```

```
Aluno aluno;

int main() {
    setlocale(LC_ALL,"portuguese");
    aluno.codigo = 0;
    strcpy(aluno.nome, "NULL");
    aluno.datNasc.dia = 0;
    aluno.datNasc.mes = 0;
    aluno.datNasc.ano = 0;
    printf("O código do aluno é: %d\n", aluno.codigo);
    printf("O nome do aluno é: %s\n", aluno.nome);
    printf("A data de nascimento do aluno é: %d / %d / %d\n", aluno.datNasc.dia,
aluno.datNasc.mes, aluno.datNasc.ano);
    printf(" \n \n");
    printf(" Digite o código do aluno: ");
    scanf("%d%*c", &aluno.codigo);
    printf(" Digite o nome do aluno: ");
    scanf("%s%*c", &aluno.nome);
    printf(" Digite o dia do nascimento do aluno: ");
    scanf("%d%*c", &aluno.datNasc.dia);
    printf(" Digite o mês do nascimento do aluno: ");
    scanf("%d%*c", &aluno.datNasc.mes);
    printf(" Digite o ano do nascimento do aluno: ");
    scanf("%d%*c", &aluno.datNasc.ano);
    printf(" \n O código do aluno é: %d ", aluno.codigo);
    printf(" \n O nome do aluno é: %s ", aluno.nome);
    printf(" \n A data de nascimento do aluno é: %d / %d / %d ",
aluno.datNasc.dia, aluno.datNasc.mes, aluno.datNasc.ano);
    printf(" \n \n");
    system("pause");

    return 0;
}
```

Neste exemplo, `Data` é declarada antes de `Aluno` e é acessada por `aluno.datNasc.dia`, `aluno.datNasc.mes`, `aluno.datNasc.ano`.

Atenção: É necessário declarar estruturas externas antes de utilizá-las dentro de outras estruturas. Caso contrário, o compilador gerará um erro.

Array de Structs

Um **array de structs** é uma estrutura de dados que armazena uma sequência de objetos do mesmo tipo, onde cada objeto é uma estrutura. No exemplo apresentado, cada posição do vetor armazena informações diferentes, como nome do aluno, nome da disciplina e nota (tipos string e float).

Arrays de structs são úteis para criar **listas** e podem ser aplicados em tarefas simples, como **leitura repetida de dados** em cadastros.

Declaração de Array de Structs:

Para declarar um **array de structs**, define-se uma estrutura, como no exemplo:

```
typedef struct {
    char nome[200];
    char disciplina[100];
    float nota;
} Aluno;

Aluno aluno_nota[10];
```

Aqui, na linha 7, é criado um **vetor de 10 posições** para armazenar o nome do aluno, a disciplina e a nota, sendo cada posição do vetor uma estrutura do tipo `Aluno`.

Inicialização de Array de Struct:

Para inicializar um **array de structs**, é necessário atribuir valores padrão aos membros de cada estrutura. No exemplo, isso é feito utilizando o loop `for`:

```
for(i = 0; i < 10; i++) {
    strcpy(aluno_nota[i].nome, "NULL");
    strcpy(aluno_nota[i].disciplina, " ");
    aluno_nota[i].nota = 0.0;
}
```

Aqui, a função `strcpy` é usada para copiar valores para os membros do tipo string (`nome` e `disciplina`), enquanto o valor `0.0` é atribuído ao membro `nota`.

População de Array de Struct:

Após a inicialização do array de structs, os dados podem ser inseridos usando um laço `for`. Cada elemento do array é preenchido com informações fornecidas pelo usuário. A sintaxe para armazenar os dados é:

```
&nome_vetor_struct[indice].nome_membro_struct;
```

O código **`&nome_vetor_struct[indice].nome_membro_struct;`** representa a obtenção do **endereço de memória** de um membro de uma estrutura que está contida em um vetor de estruturas em C.

Exemplo de código para armazenar as informações:

```
for (i = 0; i < 10; i++) {
```

```
    printf("Digite nome do aluno: ");
    scanf("%s%*c", &aluno_nota[i].nome);
    printf("Digite a disciplina do aluno: ");
    scanf("%s%*c", &aluno_nota[i].disciplina);
    printf("Digite a nota do aluno: ");
    scanf("%f%*c", &aluno_nota[i].nota);
}
```

Este código lê e armazena o nome, a disciplina e a nota de cada aluno no array `aluno_nota`.

Impressão de Array de Struct:

Para imprimir os valores de um array de structs, utiliza-se um laço `for` para percorrer as posições do array. A sintaxe para imprimir um membro de cada struct é:

```
printf("\nTEXTO %_", nome_vetor_struct[indice].nome_membro_struct);
```

Substitua `%_` pelo tipo correspondente (ex.: `%d` para inteiros, `%.2f` para floats). Exemplo de código para imprimir os dados do array `aluno_nota`:

```
for(i = 0; i < 10; i++) {
    printf("\nO nome do aluno é: %s", aluno_nota[i].nome);
    printf("\nA disciplina do aluno é: %s", aluno_nota[i].disciplina);
    printf("\nA nota do aluno é: %.2f", aluno_nota[i].nota);
}
```

Este código imprime o nome, a disciplina e a nota de cada aluno no array.

Busca em Array de Struct:

Para buscar um elemento em um array de structs, como um nome, utiliza-se um laço `for` para percorrer o vetor e comparar o valor procurado com os membros da struct. No caso de strings, a função `strcmp` é utilizada para comparar a string digitada com as strings armazenadas.

Exemplo de código para busca de nome:

```
printf("\nDigite um nome: ");
scanf("%s%*c", nome);

for(i = 0; i < 10; i++) {
    if(strcmp(nome, aluno_nota[i].nome) == 0) {
        printf("\nRegistro encontrado!");
        posicao = i;
    } else {
        posicao = -1;
    }
}

if(posicao == -1) {
    printf("\nRegistro não encontrado!");
} else {
    printf("\nRegistro encontrado: ");
    printf("\nO nome do aluno é: %s", aluno_nota[posicao].nome);
    printf("\nA disciplina do aluno é: %s", aluno_nota[posicao].disciplina);
```

```
    printf("\nA nota do aluno é: %.2f", aluno_nota[posicao].nota);
}
```

Explicação:

1. O nome é lido e armazenado na variável `nome`.
2. O laço `for` percorre o array e compara cada `nome` da struct com a string digitada pelo usuário usando `strcmp`.
3. Se o nome for encontrado (resultado de `strcmp` igual a 0), a posição é salva e exibida. Caso contrário, é retornado que o registro não foi encontrado.

Exemplo Prático de C:
O código demonstra como utilizar structs para armazenar e manipular informações de alunos, incluindo dados pessoais e de nascimento.

Estruturas:

1. **Data:** Representa a data de nascimento (dia, mês, ano).
2. **Aluno:** Contém o código do aluno, nome e data de nascimento (do tipo `Data`).

Fluxo do programa:

1. **Declaração e Inicialização:**
    - Declara-se um array de 5 structs do tipo `Aluno` (variável `aluno[5]`).
    - Inicializa-se os campos de cada struct com valores padrão (código 0, nome "NULL", e data de nascimento 0/0/0).
2. **Exibição Inicial:**
    - O programa imprime os valores padrão de cada aluno.
3. **Entrada de Dados:**
    - O programa solicita ao usuário os dados para cada aluno (código, nome, e data de nascimento), utilizando `scanf`.
4. **Exibição Final:**
    - Após a entrada de dados, o programa imprime as informações inseridas para cada aluno.

Código Completo:

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

typedef struct {
    int dia;
    int mes;
    int ano;
} Data;

typedef struct {
    int codigo;
    char nome[200];
    Data datNasc;
```

```
} Aluno;

Aluno aluno[5];
int i = 0;

int main() {
    setlocale(LC_ALL, "portuguese");

    // Inicialização dos dados
    for(i = 0; i < 5; i++) {
        aluno[i].codigo = 0;
        strcpy(aluno[i].nome, "NULL");
        aluno[i].datNasc.dia = 0;
        aluno[i].datNasc.mes = 0;
        aluno[i].datNasc.ano = 0;
    }

    // Exibição dos dados iniciais
    for(i = 0; i < 5; i++) {
        printf("\n ======================================");
        printf("\n O código do aluno é: %d ", aluno[i].codigo);
        printf("\n O nome do aluno é: %s ", aluno[i].nome);
        printf("\n A data de nascimento do aluno é: %d / %d / %d ",
               aluno[i].datNasc.dia, aluno[i].datNasc.mes,
aluno[i].datNasc.ano);
        printf("\n");
    }

    // Entrada de dados dos alunos
    for(i = 0; i < 5; i++) {
        printf("\n ======================================");
        printf("\n Digite o código do aluno: ");
        scanf("%d%*c", &aluno[i].codigo);
        printf("\n Digite o nome do aluno: ");
        scanf("%s%*c", aluno[i].nome);
        printf("\n Digite o dia do nascimento do aluno: ");
        scanf("%d%*c", &aluno[i].datNasc.dia);
        printf("\n Digite o mês do nascimento do aluno: ");
        scanf("%d%*c", &aluno[i].datNasc.mes);
        printf("\n Digite o ano do nascimento do aluno: ");
        scanf("%d%*c", &aluno[i].datNasc.ano);
    }

    // Exibição dos dados inseridos
    for(i = 0; i < 5; i++) {
        printf("\n ======================================");
        printf("\n O código do aluno é: %d ", aluno[i].codigo);
        printf("\n O nome do aluno é: %s ", aluno[i].nome);
        printf("\n A data de nascimento do aluno é: %d / %d / %d ",
               aluno[i].datNasc.dia, aluno[i].datNasc.mes,
aluno[i].datNasc.ano);
        printf("\n");
    }

    system("pause");
    return(0);
}
```

Explicação:

- **Inicialização:** O array de alunos é preenchido com valores padrão antes de coletar as informações.

- **Entrada:** O programa coleta dados para cada aluno (código, nome, data).
- **Exibição:** Depois de inseridos, os dados são exibidos novamente.

Modularização é o processo de dividir um sistema em partes autônomas que podem ser combinadas para formar um todo. Na computação, é usada para dividir programas em funções independentes, facilitando a comunicação entre elas.

A **história** da modularização segue um paralelo com a **Revolução Industrial** do século XIX, que introduziu a **linha de montagem**, onde o trabalho era dividido em etapas, tornando a produção mais eficiente. Antes disso, produtos eram manufaturados inteiramente por um único artesão.

Na computação, o conceito de modularização começou com o **ENIAC**, o primeiro computador digital, que não tinha software, apenas hardware. Em 1950, **John Von Neumann** criou a **arquitetura de computadores**, dividindo o computador em três partes: Entrada, Processamento e Saída. Isso ajudou a modularizar o hardware e o software dos computadores.

Nos anos 1960, a programação ainda era feita de forma monolítica, gerando **redundância de código** e dificuldades de manutenção. A crise do software na década de 1960 levou à criação da **Engenharia de Software** após a Conferência da OTAN de 1968, onde práticas mais maduras para o desenvolvimento de software foram estabelecidas. A **Programação Estruturada** surgiu ao lado da Engenharia de Software, permitindo a divisão de códigos em partes especializadas, melhorando a organização e a eficiência no desenvolvimento de sistemas complexos.

Esse movimento reflete a **Revolução Industrial** da computação, com a busca por uma abordagem mais sistemática e controlada para o desenvolvimento de software.

A **modularização** é o processo de dividir um programa em subprogramas individuais para facilitar a construção de programas grandes, decompondo-os em etapas menores. O lema é “dividir para conquistar”, abordando o problema em subproblemas menores.

- **Programa principal**: é o ponto de início da execução e chama os subprogramas.
- **Subprograma**: resolve pequenos problemas e é chamado pelo programa principal ou por outro subprograma.

A modularização oferece várias vantagens:

- Evita programas grandes e difíceis de entender.
- Facilita a leitura e compreensão do código.
- Permite a criação de módulos independentes e reutilizáveis.
- Evita a repetição de código.
- Facilita a modificação rápida de trechos do programa.
- Permite o reaproveitamento de código em outros programas (bibliotecas).

As **funções** e **procedimentos** são blocos de código que executam ações específicas, sendo mais úteis e reutilizáveis quando realizam apenas uma ação. Isso permite conectar módulos para resolver problemas mais complexos.

A **programação estruturada** surgiu para tornar os programas mais claros e fáceis de entender, dividindo-os em partes menores. Inicialmente, os programas eram blocos lógicos com estruturas de controle (condição, seleção e repetição), mas com o tempo, a divisão em sub-rotinas passou a ser utilizada.

Sub-rotina: ferramenta que ajuda a evitar a repetição de código e a dividir algoritmos em partes coerentes.

- **Procedimentos**: agrupam comandos que são executados quando chamados, mas não retornam valores.
- **Funções**: semelhantes aos procedimentos, mas sempre retornam um valor.

Essas sub-rotinas se comunicam com o programa principal, permitindo maior modularização e clareza no código.

Funções são procedimentos que retornam um único valor após a execução. Exemplos comuns incluem:

- **sqrt()**: calcula a raiz quadrada.
- **scanf()**: lê um número da entrada.
- **printf()**: imprime um valor na saída.

Essas funções realizam tarefas específicas e retornam valores para o programa.

Declaração de função:
Uma função em pseudolinguagem tem o seguinte formato:

```
kotlin
```

```
funcao <nome-da-função>
    [(<declaração-de-parâmetros>)] : <tipo-de-dado>
    var
    // Declarações internas
    inicio
    // Comandos
fimfuncao
```

- Toda função deve ter um tipo que define o valor de retorno.
- Parâmetros determinam o comportamento da função, mas uma função pode não ter parâmetros.
- Funções devem ser declaradas antes do programa principal. Caso contrário, a declaração deve ocorrer antes da função `main()`.
- Para declarar uma função sem implementá-la, usa-se apenas o tipo e a assinatura:

```
php
```

```
tipo nome_da_funcao(tipo <param1>, tipo <param2>, ...);
```

Por exemplo, em C:

```
arduino
```

```
void imprime(int numero);
```

Declarar funções antes do uso facilita a leitura e compreensão do código.

No exemplo abaixo, a função `soma()` em C realiza a soma de dois valores passados como parâmetros:

c

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

- O `return` define o valor de retorno da função, sendo a última instrução executada na função.
- Esse valor de retorno também é usado pelo sistema operacional para indicar o sucesso ou falha de um programa: o retorno zero indica sucesso; qualquer outro valor, falha.

A invocação de uma função ocorre ao atribuir seu valor a uma variável:

c

```
variavel = funcao(parametros);
```

Também é possível invocar funções diretamente em expressões, como em:

c

```
printf("Soma de x e y: %d\n", soma(x, y));
```

Neste caso, `soma(x, y)` é chamada durante a execução de `printf`, e o resultado da soma é exibido na tela. Os valores passados como parâmetros são copiados para os parâmetros da função, que pode manipulá-los, mas esses valores originais não são alterados.

Tipo `void`: Usado em funções ou variáveis que não retornam ou armazenam valores definidos, representando o "nada".

Procedimentos

Procedimentos são estruturas que agrupam comandos e são executados quando chamados. Em C, procedimentos são funções do tipo `void`, que não retornam valor.

Declaração de um procedimento: O formato geral é:

c

```
procedimento <nome-do-procedimento> [(parâmetros)]
var
// Declarações Internas
inicio
// Comandos
fimprocedimento
```

Exemplo: O procedimento `void imprime(int numero)` imprime o número passado como parâmetro:

c

```
void imprime(int numero) {
    printf("Número %d\n", numero);
}
```

Ao ignorar o valor de retorno de uma função, ela se comporta como um procedimento.

Invocando um procedimento

Para invocarmos um procedimento, devemos utilizá-lo como utilizaríamos qualquer outro comando, ou seja:

```
procedimento (parametros);
```

Em C, as funções predefinidas estão nas bibliotecas (arquivos header ".h"), que devem ser incluídas no início do programa com `#include`. Abaixo, algumas bibliotecas padrão:

- **stdio.h**: Funções de entrada/saída para dispositivos padrão (stdin, stdout) e arquivos.
- **stdlib.h**: Funções de entrada/saída e conversão de números para strings.
- **ctype.h**: Funções para manipulação de caracteres.
- **time.h**: Funções para datas e horários.
- **dos.h**: Funções para acessar as interrupções da BIOS e MS-DOS.
- **string.h**: Funções para manipulação de strings.
- **math.h**: Funções matemáticas.

Funções de E/S padrão

As funções de entrada/saída (E/S) padrão em C permitem operações básicas de E/S. Ao executar um programa em C, o sistema operacional abre três arquivos: **stdin** (entrada padrão, teclado), **stdout** (saída padrão, vídeo) e **stderr** (erro padrão, vídeo). Para usar essas funções, é necessário incluir `#include <stdio.h>` no início do programa.

Função `printf` para Formatação de Saída em C

A função `printf` é usada em C para imprimir dados formatados na saída padrão (geralmente a tela). Ela permite a exibição de valores utilizando uma variedade de especificações de formato para controle de como os dados são apresentados.

Sintaxe Básica

c

```
printf("format_string", arg1, arg2, ...);
```

- **format_string**: Inclui caracteres comuns, que são copiados diretamente para a saída, e especificações de formato que determinam como os argumentos devem ser exibidos.
- **arg1, arg2, ...**: São variáveis cujos valores são impressos de acordo com as especificações na `format_string`.

Especificações de Formato

Cada especificador de formato começa com `%` e é seguido por modificadores opcionais que controlam detalhes da impressão:

1. **Flags**:
    - `-`: Alinha o valor à esquerda.
    - `+`: Exibe o sinal `+` ou `-` antes do valor.
    - (espaço): Adiciona um espaço antes de números positivos.
    - `0`: Preenche com zeros à esquerda.
2. **Largura (width)**:
    - `<n>`: Define o número mínimo de caracteres. Se o valor for menor, completa com espaços.
    - `0<n>`: Completa com zeros em vez de espaços se o valor for menor.
3. **Precisão**:
    - `.0`: Para floats, omite o ponto decimal.
    - `.<n>`: Define o número de casas decimais para números.
4. **Tamanho (size)**:
    - `l`: Modifica para LONG (para números maiores).
    - `h`: Modifica para SHORT (para números menores).
5. **Tipos de Dados**:
    - `d` ou `i`: Inteiro decimal.
    - `o`: Inteiro octal sem sinal.
    - `x` ou `X`: Inteiro hexadecimal, usando letras `a-f` ou `A-F`.
    - `u`: Inteiro decimal sem sinal.
    - `c`: Caractere único.
    - `s`: String (cadeia de caracteres).
    - `f`: Float ou double em notação decimal.
    - `e` ou `E`: Float ou double em notação científica.
    - `p`: Ponteiro.
    - `%`: Exibe o caractere `%`.

Exemplo de Uso

`c`

```
int x = 10, y = 5;
printf("Soma de x e y: %d\n", x + y);
```

Nesse exemplo, `printf` exibe o texto "Soma de x e y: 15", onde `%d` indica que o argumento será um número decimal (`int`). A função chama `x + y` e exibe o resultado `15` na saída padrão.

Erros Comuns

Se `printf` for invocado sem argumentos quando são necessários, ocorre o erro: *too few arguments to function 'printf'*.

O `printf` é uma ferramenta versátil para formatação de dados em C, permitindo um controle detalhado sobre a saída, essencial para exibições precisas e personalizadas.

Exemplos de Uso da Função `printf`

A função `printf` é usada para exibir dados formatados, controlando o tipo e a formatação de cada valor impresso. Aqui estão exemplos de uso e suas explicações:

1. **Formatando Ponto Flutuante com Casas Decimais**:

   c

   ```
   printf("\nNota do aluno: %3.2f", nota);
   ```

   - Exibe `nota` com três dígitos, sendo duas casas decimais.

2. **Imprimindo String e Valor em Hexadecimal**:

   c

   ```
   printf("Lista de itens: %s\nContador = %x", string, contador);
   ```

   - Exibe `string` como uma cadeia de caracteres e `contador` em hexadecimal.

3. **Alinhando Inteiros à Esquerda**:

   c

   ```
   printf("a: %-5D b: %-5D c: %-5D", a, b, c);
   ```

   - Alinha os valores de `a`, `b` e `c` à esquerda com 5 espaços.

Exemplo: Programa 1 (`Programa1.c`)

c

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

int main(int argc, char *argv[]) {
    int i = 123;
    printf("%d\n", i);
    return 0;
}
```

- `%d` indica que `i` será exibido como um número decimal. `\n` adiciona uma nova linha. Alterando para `%x`, `i` seria exibido como `7B` (base 16).
- **`argc`**: Contém o número de argumentos passados na linha de comando (incluindo o nome do programa).
- **`argv`**: É um array de strings contendo os argumentos.

Exemplo: Programa 2 (`Programa2.c`)

c

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

```
int main(int argc, char *argv[]) {
    double pi = 3.14;
    printf("%f\n", pi);
    return 0;
}
```

- `%f` exibe `pi` com ponto flutuante, no formato `[-] m.dddddd`.

Exemplo: Programa 3 (`Programa3.c`)

c

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

int main(int argc, char *argv[]) {
    int i = 123;
    double pi = 3.14;
    printf("%d %f\n", i, pi);
    return 0;
}
```

- Combina `%d` e `%f`, exibindo `i` em decimal e `pi` em ponto flutuante, com cada valor na formatação especificada.

Esses exemplos demonstram a versatilidade de `printf` para exibir valores em diferentes formatos, como decimal, hexadecimal e ponto flutuante, dependendo das especificações de conversão (`%d`, `%x`, `%f`).

Função `scanf` em C

A função `scanf` permite a entrada de dados formatados, lendo-os do teclado (dispositivo de entrada padrão) e armazenando-os nas variáveis especificadas. Funciona de forma análoga ao `printf`, mas para entrada.

Especificações de Formatação

Caractere	Dados de Entrada	Tipo de Argumento
`d`	Número inteiro decimal	`int *`
`i`	Inteiro	`int *`
`o`	Octal inteiro	`int *`
`u`	Inteiro decimal sem sinal	`unsigned int *`
`x`	Número inteiro hexadecimal	`int *`
`c`	Caractere	`char *`
`s`	Cadeia de caracteres (string)	`char *`
`e`, `f`, `g`	Número de ponto flutuante	`float *`
`%`	Literal `%`	

Essas especificações definem o tipo de dado que será lido e armazenado.

Sintaxe

c

```
scanf("format_string", &arg1, &arg2, ...);
```

- **format_string**: Define a formatação dos dados a serem lidos, como tipos e posições.
- **Argumentos**: São ponteiros para as variáveis onde os dados serão armazenados. É fundamental incluir & antes do nome da variável, exceto para arrays. A omissão do & resulta em um erro comum entre iniciantes.

Observação: Ao omitir argumentos em `scanf`, ocorre um erro de falta de argumentos, indicando "too few arguments to function 'scanf'".

Exemplos de Uso da Função `scanf`

A função `scanf` lê dados do teclado e os armazena em variáveis com base nas especificações de formatação.

Exemplos

1. Para ler um número de ponto flutuante e armazenar em `salario`:

   c

   ```
   scanf("%f", &salario);
   ```

2. Para ler um número inteiro e armazenar em `numero`:

   c

   ```
   scanf("%i", &numero);
   ```

Programa Exemplo (Programa4.c)

c

```
1   //Programa 4
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   int main(int argc, char *argv[]){
6       int i;
7       scanf("%i", &i);
8       printf("%i\n", i);
9       return 0;
10  }
```

Este programa usa `scanf` para ler um número inteiro do teclado e armazená-lo na variável `i`. Em seguida, `printf` exibe o valor de `i` no formato inteiro.

Comportamento da Função `scanf`

- A função `scanf` processa os dados conforme a especificação de conversão.
- Para conversão com `%d`, ela ignora caracteres até encontrar uma sequência numérica válida.

- Para `%c`, `scanf` não ignora nenhum caractere, incluindo o caractere de nova linha ("Enter").
- A função retorna ao ponto de chamada apenas após completar a conversão.

Observação: O caractere de nova linha é tratado como espaço em branco, exceto para a conversão %c.

Outras Funções de Entrada e Saída de Dados

1. **putchar**
    - Envia um único caractere para a saída padrão (tela).
    - Equivalente a: `printf("%c", carac)`.
    - Sintaxe: `putchar(carac);`
    - Exemplo: `putchar('A');` imprime "A" na tela.
2. **puts**
    - Envia uma string para a saída padrão, seguida automaticamente por um caractere de nova linha (`\n`).
    - Equivalente a: `printf("%s\n", string)`.
    - Sintaxe: `puts(string);`
    - Exemplo: `puts("Olá!");` imprime "Olá!" e pula para a próxima linha.
3. **getchar**
    - Lê um único caractere da entrada padrão (teclado).
    - Equivalente a `putchar`, mas em direção oposta, ou seja, para entrada de dados.
    - Sintaxe: `[variável] = getchar();`
    - Exemplo: `ch = getchar();` lê o caractere digitado e armazena na variável `ch`.
4. **gets**
    - Lê uma string (sequência de caracteres) da entrada padrão até encontrar um caractere de nova linha (`\n`).
    - Equivalente a `puts`, mas para leitura de dados.
    - Sintaxe: `[variável] = gets();`
    - Exemplo: `gets(nome);` lê a entrada do usuário e armazena na variável `nome`.

Atenção: Para evitar problemas ao usar `getchar` e garantir que o buffer de teclado não interfira nas leituras, deve-se usar a função `fflush(stdin)` no Windows ou `_fpurge(stdin)` no Linux antes de chamar `getchar`. Isso limpa o buffer de entrada, garantindo que não haja lixo ou caracteres indesejados.

Funções de Manipulação de Arquivos

As funções de manipulação de arquivos, localizadas na biblioteca `stdio.h`, são responsáveis por realizar operações com arquivos. Para usá-las, deve-se incluir a seguinte linha no início do programa:

c

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

Função de Manipulação de Arquivos - `fopen`

A função `fopen` abre arquivos ou dispositivos (como console e impressora) para operações. Para utilizá-la, é necessário declarar um ponteiro para o arquivo do tipo `FILE`.

Declaração de Ponteiro de Arquivo:

c

```
FILE *fp;
```

- `FILE`: Tipo de dado da biblioteca `stdio.h`.
- `*fp`: Ponteiro que referencia o arquivo.

Sintaxe do `fopen`:

c

```
FILE *fopen(const char *nome_arquivo, const char *modo);
```

- **nome_arquivo**: Nome do arquivo.
- **modo**: Especifica a operação do arquivo:
  - `"r"`: Leitura.
  - `"w"`: Escrita.
  - `"a"`: Adição.
  - `"r+"`: Leitura e escrita.
  - `"w+"`: Criação para atualização.
  - `"a+"`: Adição, com leitura e escrita.

Retorno: Retorna o ponteiro para o arquivo ou `NULL` em caso de erro.

Exemplo:

c

```
fp = fopen("arquivo.txt", "r");
```

Outras Funções de Manipulação de Arquivos

1. `fgetc`
   - Lê um caractere de um arquivo. Retorna `EOF(End Of File)` ao chegar ao fim do arquivo.
   - Sintaxe: `caractere = fgetc(file_pointer);`
2. `fputc`
   - Insere um caractere no arquivo na posição atual do ponteiro. Operação inversa de `getc`.
   - Sintaxe: `fputc(caractere, file_pointer);`
3. `ungetc`
   - "Devolve" o último caractere lido ao arquivo. Apenas um caractere pode ser devolvido de cada vez.

- Sintaxe: `ungetc(caractere, file_pointer);`

4. `fprintf`
    - Realiza uma saída formatada para um arquivo. Equivalente ao `printf`, mas direcionado para arquivos. O primeiro argumento é o ponteiro para o arquivo onde os dados serão escritos.
    - Sintaxe: `fprintf(file_pointer, "controle", arg1, arg2, ...);`
5. `fscanf`
    - Realiza uma entrada formatada a partir de um arquivo. Funciona como o `scanf`, mas para leitura de arquivos. O primeiro argumento é o ponteiro para o arquivo de onde os dados serão lidos.
    - Sintaxe: `fscanf(file_pointer, "controle", arg1, arg2, ...);`
6. `fclose`
    - Fecha um arquivo aberto e libera os recursos associados a ele.
    - Sintaxe: `fclose(file_pointer);`

Outras Funções de Manipulação de Arquivos

- `fflush()`
    - Descarrega o buffer de um arquivo, forçando a escrita imediata dos dados no arquivo.
- `fgets()`
    - Lê uma linha de texto (string) do arquivo. A leitura é feita até um caractere de nova linha ou até atingir o limite de tamanho da string.
- `fputs()`
    - Insere uma string no arquivo. A string é escrita até o caractere de nova linha `\n`.
- `fread()`
    - Lê um bloco de dados do arquivo. Utilizada para ler dados binários em blocos de tamanho específico.
- `fwrite()`
    - Escreve um bloco de dados no arquivo, útil para gravações binárias.
- `fseek()`
    - Reposiciona o ponteiro do arquivo para uma posição específica, permitindo navegação avançada dentro do arquivo.
- `rewind()`
    - Reseta o ponteiro de leitura/escrita para o início do arquivo.
- `ftell()`
    - Retorna a posição atual do ponteiro no arquivo. Usado para verificar a posição do cursor ou controlar o fluxo de leitura/escrita.

Funções de Manipulação de Tipos de Dados

Essas funções permitem o tratamento de caracteres, verificando características como se o caractere é ASCII, numérico, maiúsculo ou minúsculo.

Declaração

Para utilizar essas funções, é necessário incluir a biblioteca:

c

```
#include <ctype.h>
```

Funções Comuns

- **`isupper(c)`**
  Verifica se o caractere é maiúsculo. Retorna diferente de zero se for, e zero caso contrário. Função inversa: `islower(c)` (verifica se é minúsculo).
- **`isalpha(c)`**
  Verifica se o caractere é alfabético (letra). Funciona de forma semelhante ao `isupper`.

Funções Análogas

- **`isdigit(c)`** – Verifica se o caractere é um dígito numérico.
- **`isspace(c)`** – Verifica se o caractere é um espaço em branco.
- **`isascii(c)`** – Verifica se o caractere é um valor ASCII válido.
- **`isprint(c)`** – Verifica se o caractere pode ser impresso.
- **`tolower(c)`**
  Converte o caractere de maiúsculo para minúsculo. Função inversa: `toupper(c)` (converte de minúsculo para maiúsculo).

Funções de Manipulação de String

Essas funções são usadas para manipular strings, como copiar, comparar e concatenar.

Declaração

Para utilizar essas funções, é necessário incluir a biblioteca:

c

```
#include <string.h>
```

Funções Comuns

- **`strlen(string)`**
  Retorna o tamanho de uma string.
- **`strcpy(s, t)`**
  Copia a string `t` para a string `s`.

- **`strcmp(s, t)`**
  Compara as duas strings `s` e `t`.
- **`strcat(s, t)`**
  Concatena a string `t` ao final da string `s`.

Introdução

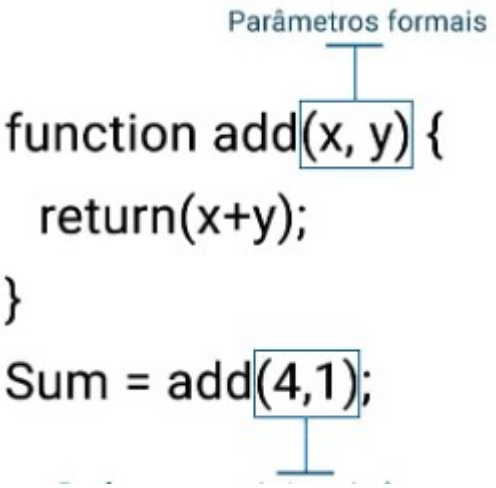
No módulo anterior, estudamos funções predefinidas como `printf`, `scanf`, `getchar`, `putchar`, entre outras, que fazem parte da biblioteca padrão C. Essas funções recebem parâmetros e são invocadas no programa principal com argumentos conforme a sintaxe definida.

Exemplo

A função `scanf` lê dados de uma fonte externa. Sua sintaxe é `scanf("expressão de controle", lista de argumentos)`, onde os argumentos especificam o formato dos valores e os endereços das variáveis que receberão esses valores.

Definições

- **Parâmetro:** Variável que recebe valores passados para uma sub-rotina, alterando seu comportamento durante a execução.
- **Parâmetros formais:** Definidos na declaração da sub-rotina, servindo como variáveis locais.
- **Argumentos (ou parâmetros reais/atuais):** Valores ou variáveis passadas para os parâmetros formais na invocação da sub-rotina.



Exemplo: Na função `add(x, y)` que soma dois inteiros, `x` e `y` são os parâmetros formais, e ao invocar a função com `add(4, 1)`, os valores `4` e `1` são os argumentos passados para a função.

Passagem de Parâmetros

Quando uma sub-rotina é chamada, é necessário passar valores para seus parâmetros, e esses valores devem corresponder ao tipo e à quantidade de parâmetros esperados. A passagem de parâmetros pode ser feita de duas formas: **por valor** ou **por referência**.

Passagem por Valor

Na passagem por valor, a sub-rotina recebe uma cópia do valor do argumento passado. Ou seja, a variável do programa principal não é alterada, pois o que é manipulado é apenas a cópia do valor. Isso é ilustrado no exemplo da função `troca()`, onde, ao passar `v1` e `v2` como argumentos, a troca acontece dentro da sub-rotina, mas as variáveis originais (`v1` e `v2`) no programa principal não são alteradas.

- **Exemplo (Programa 5)**:

- O programa principal invoca `troca(v1, v2)`. Os valores de `v1` e `v2` (5 e 10) são copiados para os parâmetros `x` e `y` da função `troca()`.
- Dentro de `troca()`, as variáveis `x` e `y` são trocadas, mas isso não afeta as variáveis `v1` e `v2` no programa principal, já que foi passada uma cópia de seus valores.

Conceito: A passagem por valor entrega uma cópia (como uma "xerox") dos valores originais, sem modificar o valor original da variável no programa principal.

Passagem por Referência

Na passagem por referência, ao invés de passar o valor da variável, é passado o endereço de memória da variável, permitindo que a sub-rotina altere o valor da variável original diretamente. Isso é feito através do uso de **ponteiros**, que armazenam o endereço de memória de uma variável.

- **Exemplo (Programa 6)**:
  - A sub-rotina `troca()` agora é chamada com os endereços de `v1` e `v2` (usando o operador &), passando esses endereços para os ponteiros `*x` e `*y`.
  - Dentro de `troca()`, os valores armazenados nos endereços de `v1` e `v2` são alterados diretamente, pois a função manipula o conteúdo da memória apontada pelos ponteiros, e essas alterações são refletidas no programa principal.

```c
//Programa 6
#include < stdio.h >
void troca(int *x, int *y) {
    int aux;
    if(x != NULL && y != NULL){ //se endereç
    aux = *x; //faz a troca
    *x = *y;
    *y = aux;
    }
}

int main(){
    int v1=5, v2=10;
    troca(&v1, &v2);
    printf("v1 = %d e v2 = %d\n", v1, v2);
}
```

Conceito: Na passagem por referência, a função recebe o endereço de memória da variável e pode alterá-la diretamente, alterando assim o valor original da variável.

Considerações Importantes:

- **Passagem por Valor**: Só é passada uma cópia do valor. Alterações não afetam a variável original.
- **Passagem por Referência**: O endereço da variável é passado, permitindo que a função altere diretamente a variável original.

Além disso, na passagem por referência, deve-se passar variáveis que contenham endereços válidos, e o tipo dos parâmetros na sub-rotina deve ser ponteiros, como `int *x` e `int *y`. O uso de ponteiros exige o operador & para obter o endereço de memória da variável e o operador * para acessar o valor no endereço.

A passagem de parâmetros, seja por valor ou por referência, é destruída ao final da execução da sub-rotina, mas na passagem por referência, o que é destruído é a referência (o ponteiro), não a variável original.

Protótipos de Sub-rotinas

Para utilizar sub-rotinas (funções e procedimentos) no programa, é necessário declarar **protótipos** ou **assinaturas** dessas sub-rotinas. Elas representam a primeira linha da definição de uma função ou procedimento e devem ser posicionadas antes do uso da sub-rotina no código.

- **Protótipos/Assinaturas**: São declarações que incluem o tipo de retorno, o nome da função ou procedimento e os tipos de seus parâmetros. Elas servem para informar ao compilador a existência da sub-rotina antes de sua utilização.

Exemplo de sintaxe:

```
c

tipo nome_da_função(tipo parâmetro1, ..., tipo parâmetroN) {
    comandos;
    return;
}

void nome_do_procedimento(tipo parâmetro1, ..., tipo parâmetroN) {
    comandos;
}
```

Essas assinaturas são geralmente colocadas no início do arquivo-fonte.

- **Exemplo Prático (Programa 7)**:

```
c

//Programa 7
#include <stdio.h>

// Assinatura da função
int soma(int x, int y);

int main(void) {
    int num1, num2, num3;
    printf("Digite primeiro numero: ");
    scanf("%d", &num1);
    printf("Digite segundo numero: ");
    scanf("%d", &num2);
    num3 = soma(num1, num2); // Chamada da função
    printf("A soma de: %d + %d = %d\n", num1, num2, num3);
    return(0);
}

// Definição da função soma
int soma(int x, int y) {
    return (x + y);
}
```

Explicação do Código:

1. **Assinatura**: Na linha 4, temos a assinatura da função `soma` (`int soma(int x, int y);`). A assinatura informa ao compilador que a função `soma` receberá dois parâmetros inteiros e retornará um inteiro.
2. **Programa Principal**: No `main()`, as variáveis `num1` e `num2` são lidas através do `scanf`. Depois, a função `soma` é chamada na linha 12, passando `num1` e `num2` como argumentos. O resultado da soma é armazenado em `num3` e exibido na tela.
3. **Definição da Função**: A função `soma` é definida na linha 18, onde realiza a operação de soma e retorna o valor.

Esse uso de assinatura permite que a função `soma` seja chamada antes de sua definição no código, já que o compilador conhece sua assinatura e sabe como ela deve ser utilizada.

Passagem de Vetores

Em C, a passagem de vetores para funções é sempre feita por referência, ou seja, ao passar um vetor, estamos passando o endereço do primeiro elemento do vetor, não os valores individuais.

Sintaxe de passagem de vetor:

```
c

tipo nome[ ];
```

- **Tipo**: tipo dos elementos do vetor.
- **Nome**: nome do vetor.
- **[ ]**: Indica que a variável é um vetor.

A dimensão do vetor não precisa ser especificada ao passar para uma função, pois o que importa é o tipo dos seus elementos, não o tamanho.

Exemplo – Programa 8

O programa a seguir ilustra como passar um vetor para uma função usando um ponteiro. O ponteiro representa o endereço inicial do vetor.

```
c

// Programa 8
#include <stdio.h>

void v_iniciacao(int *vet, int n) {
    int i;
    for (i = 0; i < n; i++)
        vet[i] = 0;
}

void v_imprime(int *vet, int n) {
    int i;
    for (i = 0; i < n; i++)
        printf(" %d - ", vet[i]);
    printf("\n");
}

int main(void) {
    int vet[10], i;
    v_iniciacao(vet, 10); // Inicializa o vetor com zeros
    printf("Impressao do vetor antes da atribuicao:\n");
    v_imprime(vet, 10); // Imprime os valores do vetor
    for (i = 0; i < 10; i++) {
        vet[i] = i; // Atribui valores de 0 a 9 ao vetor
    }
    printf("Impressao do vetor apos a atribuicao:\n");
    v_imprime(vet, 10); // Imprime o vetor após a atribuição
    return(0);
}
```

Explicação:

1. **Função `v_iniciacao`**: Inicializa o vetor com zero.
2. **Função `v_imprime`**: Imprime os elementos do vetor.
3. **No `main`**:

- O vetor `vet` é declarado e inicializado com a função `v_iniciacao`.
- Em seguida, ele é modificado (valores de 0 a 9) e impresso novamente com a função `v_imprime`.

A passagem do vetor é feita através do ponteiro, e qualquer modificação dentro das funções afeta o vetor original.

Introdução às Variáveis e Escopo

Em programação, as variáveis são associadas aos seus valores em tempo de execução, e o escopo determina onde essas variáveis podem ser acessadas no código. Em C, as variáveis podem ser classificadas em três tipos, dependendo do local de declaração:

1. **Variáveis Globais**: Declaradas fora das sub-rotinas, podem ser acessadas por qualquer parte do programa, incluindo todas as funções.
2. **Variáveis Locais**: Declaradas dentro de uma sub-rotina, seu escopo é restrito à sub-rotina onde foram declaradas.
3. **Parâmetros Formais**: Declarados na definição das sub-rotinas, são conhecidos apenas dentro da sub-rotina e recebem valores passados externamente.

Esses três tipos de variáveis têm escopos e visibilidades diferentes, o que influencia como elas podem ser usadas no programa.

Variáveis Locais e Globais

1. **Variáveis Locais**:
   - Declaradas dentro de sub-rotinas.
   - Só podem ser usadas na sub-rotina onde são declaradas.
   - Existência limitada à execução da sub-rotina.

   Exemplo:

   ```
   c

   //Programa 9 - soma dois numeros
   #include <stdio.h>
   int main()
   {
       int num1, num2, num3;
       num1 = 10;
       num2 = 20;
       num3 = num1 + num2;
       printf("%d + %d = %d", num1, num2, num3);
       return(0);
   }
   ```

2. **Variáveis Globais**:
   - Declaradas fora das sub-rotinas.
   - Acessíveis por qualquer sub-rotina no programa, mesmo em módulos diferentes.
   - Mantêm seu valor durante toda a execução do programa.

   Exemplo:

```
c

//Programa 10 - soma dois numeros
#include <stdio.h>
int num3;
int main()
{
    int num1, num2;
    num1 = 10;
    num2 = 20;
    num3 = num1 + num2;
    printf("%d + %d = %d", num1, num2, num3);
    return(0);
}
```

3. **Conflito entre Variáveis Locais e Globais**:
    - Variáveis locais podem ter o mesmo nome que variáveis globais.
    - Dentro de uma sub-rotina, a variável local sobrepõe a variável global com o mesmo nome.

    Exemplo:

```
c

//Programa 11 - imprime o valor da variável num
#include <stdio.h>
int num;
num = 20;
int main()
{
    int num;
    num = 10;
    printf("O valor da variável num: %d", num);
    return 0;
}
```

Parâmetros Formais

- **Definição**: Parâmetros formais são variáveis locais de uma sub-rotina, usadas como entradas para a função.
- **Comportamento**: Quando se passa parâmetros para uma sub-rotina, são passadas cópias das variáveis originais, o que significa que os parâmetros formais funcionam de forma independente das variáveis que foram passadas.
- **Alteração**: É possível alterar o valor de um parâmetro formal dentro da sub-rotina, mas essa alteração não afeta a variável original.

Exemplo:

```
c

//Programa 12: soma dois números
#include <stdio.h>
//variável global
int num3;

//Função soma
int soma(int x, int y){
    int v_soma;
    v_soma = x + y;
    return(v_soma);
```

```
}

int main ()
{
    int num1, num2;
    num1 = 10;
    num2 = 20;
    num3 = soma(num1, num2);
    printf("%d + %d = %d", num1, num2, num3);
    return 0;
}
```

- **Observações**:
    - A função `soma()` recebe dois parâmetros formais (`x` e `y`), que são variáveis locais dentro da função.
    - A variável `num3` recebe o valor retornado pela função.
    - Se houver variáveis globais com o mesmo nome de variáveis locais, a versão local será a manipulada na sub-rotina, enquanto a global será acessada fora dela.

Escopo de Variáveis

- **Escopo**: O escopo de uma variável depende de onde ela é declarada no código.
- **Variáveis globais**: Declaradas fora das funções, após as bibliotecas. São acessíveis em todo o programa.
- **Variáveis locais**: Declaradas dentro de uma função. Só podem ser usadas dentro dessa função.
- **Parâmetros formais**: Declarados nas funções como variáveis locais. Sua validade está restrita à função onde foram definidos.

Exemplo (Programa 13):

```
c

//Programa 13 – Declaração de variáveis locais, globais e parâmetros formais
#include <stdio.h>
#include <conio.h>

//declaração de variáveis globais
void funcao1(variáveis locais de parâmetros)
{
    // declaração das variáveis locais da funcao1
    return;
}

// ---------------- Função main() -------------------
int main(void)
{
    //declaração das variáveis locais da main()
    return(0);
}
// -------------------------------------------------
```

- **Resumo**: As variáveis globais estão fora das funções e são acessíveis em todo o programa. As variáveis locais são específicas a cada função, e os parâmetros formais são locais a cada função que os recebe.

Introdução às Estruturas de Dados

Estruturas de dados organizam dados e suas operações, permitindo manipulação eficiente. São essenciais para profissionais de TI, sendo aplicáveis a qualquer linguagem de programação, embora o conteúdo seja apresentado com a linguagem C.

- **Estruturas de dados**: Conceitos teóricos que podem ser implementados em diferentes linguagens.
- **Objetivo**: Compreender a teoria por meio da prática na linguagem C.
- **Abordagem**:
    1. Conceitos básicos de manipulação de memória.
    2. Definição de listas, pilhas e filas como exemplos de estruturas de dados.
    3. Construção de uma visão abrangente para avançar para conceitos mais complexos.

Alocação Sequencial e Listas

A alocação sequencial é a técnica de armazenar dados em posições contíguas(adjacentes) de memória. Para utilizá-la, o programa deve informar previamente o espaço necessário, que pode ser definido em tempo de compilação ou execução. Em linguagens de programação de alto nível, essa alocação é representada por **vetores** ou **arrays**.

- **Exemplo em C**:
    - Código: `int vetor[10];` reserva espaço para 10 elementos inteiros.
    - O compilador solicita ao sistema operacional o espaço adequado, considerando o tipo de dado e a arquitetura da memória (por exemplo, um tipo `int` de 16 bits em uma memória de 8 bits requer 20 posições de memória).

```
1: [...]
2: int vetor [ 10 ];
3: int a = 50;
4: vetor [ 3 ] = a;
5: [...]
```

- **Acesso aos dados**: Os elementos de um vetor são acessados diretamente pelo índice, que é calculado a partir do endereço do primeiro elemento, permitindo operações rápidas de leitura e escrita.
- **Vantagens**:
    - Acesso eficiente aos dados devido à disposição sequencial e contígua na memória.
- **Desvantagens**:
    - Operações como remoção ou inserção de elementos são mais difíceis, pois a sequência de posições de memória precisa ser mantida. Isso torna a alocação sequencial menos flexível para modificações dinâmicas.
    - Para essas operações, é preferível utilizar **alocação dinâmica**, que permite maior flexibilidade na gestão da memória.

A alocação sequencial é importante para entender como estruturas de dados como listas, pilhas e filas funcionam, além de proporcionar uma base para o estudo de alocação dinâmica, que resolve algumas das limitações dessa abordagem.

Conceitos e Operações em Listas Lineares Genéricas

- **Listas Lineares** são estruturas de dados que armazenam um conjunto de elementos relacionados entre si. Cada elemento pode ser um dado simples ou complexo, com campos distintos. Um desses campos pode ser utilizado como **chave de busca**, um identificador único para indexar os elementos.
- **Listas Ordenadas**: Se os elementos são organizados pela chave de busca, a lista é chamada de **ordenada**. Caso contrário, é **não ordenada**. A chave de busca deve ser única e correlacionada com o elemento.

Operações em Listas Lineares

As operações básicas em listas lineares incluem:

- **Inserção**: Adicionar um novo elemento na lista.
- **Remoção**: Excluir um elemento da lista.
- **Busca**: Encontrar um elemento na lista com base na chave.

Casos Especiais de Listas Lineares

- **Deque**, **Pilha** e **Fila** são variações de listas lineares que diferem na forma como as operações de inserção e remoção são realizadas.

Esses conceitos serão aplicados e detalhados posteriormente, com o uso de **vetores de tamanho ilimitado** para implementação didática das listas.

Algoritmo de Busca

O **Algoritmo de Busca** percorre a lista a partir do início, verificando cada elemento até encontrar o item desejado ou atingir o final da lista.

- **Passos**:
  1. Verifica se a lista tem elementos (`n > 0`).
  2. Percorre a lista de 1 até `n`, comparando a chave de cada elemento.
  3. Se a chave for encontrada, retorna o índice.
  4. Se o elemento não for encontrado, retorna `n + 1`.

```
1: int buscar ( chave )
2:      se n > 0
3:              para i = 1 até i <= n
4:                      se Lista [ i ].chave == chave
6:                      retornar i
7:      retornar n + 1
```

- **Pior caso**: Quando o elemento está na última posição ou não está na lista, necessitando percorrer toda a lista.

Algoritmo de Inserção

A **inserção** de um novo elemento envolve duas etapas principais:

1. **Verificação**: Antes de inserir, o algoritmo verifica se o elemento já existe na lista, utilizando a busca. Se o elemento não for encontrado (retorno maior que `n`), pode ser inserido.
2. **Inserção**: O novo elemento é inserido após a última posição ocupada da lista, e o número de elementos (`n`) é incrementado.

- **Passos**:
    1. Se o elemento não for encontrado (verificado com a busca), insira o novo elemento na posição `n + 1`.
    2. Incrementa `n` e retorna `1` para indicar sucesso.
    3. Se o elemento já estiver na lista, retorna `-1`.

```
1: int inserir ( novo_elemento )
2:      se busca ( novo_elemento.chave ) == n + 1
3:              Lista [ n + 1 ] == novo_elemento
4:              n = n + 1
5:              retornar 1
6:      senão retornar -1
```

Esse algoritmo se aplica a listas **não ordenadas**.

Algoritmo de Remoção

O **algoritmo de remoção** realiza as seguintes etapas:

1. **Busca**: Primeiro, busca-se o elemento a ser removido. Se não encontrado, retorna erro.
2. **Deslocamento**: Se o elemento for encontrado, os elementos subsequentes são deslocados uma posição para a esquerda para preencher o espaço vazio, começando da posição do elemento removido.
3. **Atualização**: O número de elementos (`n`) é decrementado.

- **Passos**:
    1. Se a lista não estiver vazia, busca-se o elemento (`chave`).
    2. Se encontrado, todos os elementos após o elemento removido são deslocados.
    3. Decrementa `n` e retorna sucesso.
    4. Se o elemento não for encontrado, retorna `-1`. Se a lista estiver vazia, retorna erro.

```
1:      int remover ( chave )
2:              se n > 0
3:                      int i = busca ( chave )
4:                      se i < n + 1
5:                              para a = i até a < n
6:                                      Lista [ i ] = Lista [ i + 1 ]
7:                                      n = n -1
8:                      senão retornar -1
9:              senão retornar "Erro: lista vazia"
```

Considerações Práticas

- A alocação de memória é **estática** (previamente definida) na teoria, mas na prática, ela pode ser **dinâmica** (alocada em tempo de execução), utilizando funções como `malloc` em C.
- O algoritmo de remoção, ao deslocar os elementos, pode gerar um **custo de desempenho** maior em listas grandes, pois manipula todos os elementos subsequentes.

Alocação Dinâmica

- **malloc**: A função `malloc` solicita ao sistema operacional a reserva de uma área contígua de memória em tempo de execução. A quantidade de memória solicitada é calculada multiplicando o tamanho do vetor pelo tamanho do tipo de dado (neste caso, `int`).
- **Funcionamento**:

  c

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

  - O ponteiro `vetor` armazena o endereço base da área de memória alocada.
  - Os índices do vetor são deslocamentos (offsets) a partir do endereço base. Por exemplo, `vetor[n]` refere-se ao valor no deslocamento `n` a partir de `vetor`.
- **Determinando o Tamanho**: A variável `tamanho_vetor` pode ser definida durante a execução do programa, tornando a alocação dinâmica flexível.

O que o código faz:

1. **Declaração de ponteiro**:
   - `int *vetor;` declara um ponteiro para um inteiro que será usado para armazenar o endereço da memória alocada dinamicamente.
2. **Alocação de memória**:
   - `malloc(tamanho_vetor * sizeof(int))` aloca um bloco de memória suficiente para armazenar `tamanho_vetor` elementos do tipo `int`.
   - O resultado de `malloc` retorna um ponteiro `void *`, que precisa ser convertido para `int *` (essa conversão é obrigatória apenas em C++).
   - Exemplo: Se `tamanho_vetor = 5` e `sizeof(int) = 4 bytes`, `malloc` alocará 20 bytes de memória.
3. **Uso de casting**:
   - `(int *)` converte o ponteiro genérico retornado por `malloc` para um ponteiro do tipo `int`.

Vantagem de Listas Ordenadas

- Embora os algoritmos de listas possam ser aplicados tanto a listas ordenadas quanto não ordenadas, listas ordenadas oferecem vantagens adicionais, que serão exploradas mais adiante.

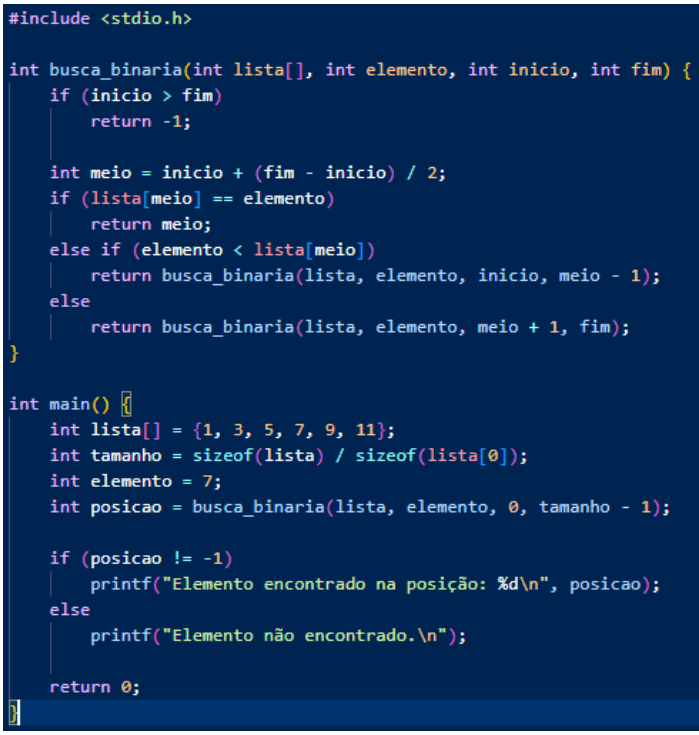
Busca Binária

A **busca binária** é um algoritmo eficiente usado em listas ordenadas. Ele funciona dividindo repetidamente a lista ao meio até encontrar o elemento ou determinar que ele não está presente.

Exemplo: Considerando a lista `[1, 3, 6, 7, 9, 12, 15, 22, 90]`, ao buscar o elemento 5, comparamos o valor com o elemento central (9). Como 9 é maior que 5, a busca continua na metade inferior da lista.

Implementação em C (Código 3):

```
c

int busca_binaria(int lista[], int elemento,
int inicio, int fim) {
    int meio = floor((fim + inicio) / 2);
    if ((inicio == fim) && (lista[meio] !=
elemento))
        return -1;
    else if (lista[meio] == elemento)
        return meio;
    else if (elemento < lista[meio])
        return busca_binaria(lista, elemento,
inicio, meio);
    else
        return busca_binaria(lista, elemento,
meio + 1, fim);
}
```



- **Passos**:
    1. Calcula o índice do meio.
    2. Se o valor no meio for igual ao elemento, retorna o índice.
    3. Se o elemento for menor que o do meio, a busca continua na metade esquerda.
    4. Se for maior, a busca segue na metade direita.
    5. Caso o elemento não seja encontrado, retorna -1.

Deque e Lista Circular

Deque (Double Ended Queue) é uma lista em que inserções e remoções ocorrem apenas nas extremidades. Isso permite que a lista cresça ou encolha pelas extremidades esquerda e direita. A alocação é sequencial, e o acesso à memória é direto. A remoção e inserção devem ser feitas nas extremidades, simplificando o gerenciamento da memória, pois não há necessidade de mover elementos internos.

Exemplo: Dada a lista `v = [a, b, r, t, c, p]`, com variáveis auxiliares `aux_esq = 0` e `aux_dir = 5` (indicando as extremidades), a remoção de `a` ou `p` é feita incrementando ou decrementando os valores de `aux_esq` ou `aux_dir`, respectivamente.

- **Inserção**: Ao inserir um elemento à esquerda, decrementa-se `aux_esq`; à direita, incrementa-se `aux_dir`. Deve-se verificar se o índice de `aux_esq` ou `aux_dir` excede os limites, evitando overflow.

Limitações: Um problema ocorre quando há espaço não utilizado entre a extremidade esquerda e direita, o que pode levar ao desperdício de memória.

Solução – Lista Circular: Ao usar uma lista circular, as extremidades direita e esquerda estão conectadas. Ao ultrapassar a extremidade direita, a lista volta à esquerda, e vice-versa. Isso otimiza o uso da memória, permitindo que o espaço de memória seja reutilizado.

No entanto, é necessário verificar os limites para evitar sobrescrições indesejadas.

Conclusão: Embora as listas circulares melhorem a utilização de memória, a capacidade do vetor continua limitada ao número de posições de memória alocadas. A escolha do mecanismo de alocação deve ser baseada nas vantagens e desvantagens de cada abordagem.

Listas Lineares Dinamicamente Encadeadas
A alocação encadeada é vantajosa ao considerar a fragmentação de memória. Com a execução de múltiplos programas, espaços de memória de tamanhos variados são deixados, dificultando a alocação de blocos contíguos. Esse fenômeno é conhecido como **fragmentação de memória** e impacta negativamente no **desempenho**, pois torna mais difícil a alocação de novas áreas de memória contíguas na heap.

Quando um programa solicita espaço para alocar um vetor de inteiros, o sistema operacional não apenas reserva um espaço, mas também verifica a tabela de alocação para encontrar um espaço suficientemente grande. O ideal é alocar o vetor em um espaço que se ajuste exatamente ao necessário, evitando o desperdício de memória. Isso mostra como a **fragmentação de memória** afeta a **performance**, pois o sistema precisa procurar por espaços adequados, o que pode aumentar o tempo de alocação.

A **alocação encadeada** resolve problemas de fragmentação de memória, permitindo que os elementos da lista não precisem estar contíguos na memória. Em vez disso, cada elemento, chamado de **nó**, armazena um ponteiro para o próximo nó, criando uma relação entre eles.
Ao contrário da alocação sequencial, onde o índice do vetor é um deslocamento direto na memória, na alocação encadeada, os elementos podem estar em posições aleatórias. Cada nó guarda o endereço do próximo elemento, permitindo o acesso à lista.

Existem dois tipos de listas encadeadas:

1. **Simplesmente encadeada**: Cada nó aponta para o próximo, permitindo movimento apenas em um sentido.
2. **Duplamente encadeada**: Cada nó possui dois ponteiros, um para o próximo e outro para o anterior, permitindo movimento nos dois sentidos.

A **alocação dinâmica** é ideal para listas encadeadas, já que o tamanho delas só é determinado em tempo de execução. Exemplos incluem listas de registros como uma agenda telefônica, onde o tamanho varia conforme a necessidade. As listas encadeadas, embora flexíveis, exigem operações específicas para criação e desalocação dos nós.

Listas Encadeadas:
Para inserir um elemento em uma lista vazia, é necessário ter uma referência para a lista, geralmente um ponteiro para o nó cabeça. Para verificar se a lista está vazia, testa-se se o campo "prox" do nó cabeça é nulo.
Se a lista estiver vazia, o novo elemento é alocado na memória e o nó cabeça passa a apontar para ele, ou seja, o campo "prox" do nó cabeça guarda o endereço do novo elemento.

Alocação de memória:
A função `calloc` é usada para reservar memória, retornando o endereço do espaço alocado. Como os nós guardam elementos e possuem ponteiros para o próximo (e o anterior, no caso de listas duplamente encadeadas), eles são tipos de dados não primitivos.

Definição de um nó em C:
A função `struct` é usada para definir um nó. Em listas duplamente encadeadas, o nó possui um campo adicional chamado "ant" para guardar o endereço do nó anterior.

Algoritmo 4 - Definição de um nó:
Em C, um nó de lista encadeada é definido usando a estrutura `struct`. O nó contém N+1 campos, sendo o último um ponteiro `prox`, que aponta para o próximo nó. Exemplo de definição de um nó genérico:
c

```
struct No {
    <tipo> campo1;
    <tipo> campo2;
    [...];
    <tipo> campoN;
    No *prox;
}
```

Para o exemplo específico de uma lista encadeada, com um campo `chave` e um ponteiro `prox`, a definição seria:

c

```
struct No {
    int chave;
    Elemento elemento;
    No *prox;
}
```

Aqui, `chave` é um campo inteiro, `elemento` pode ser outro tipo de dado e `prox` é um ponteiro para o próximo nó.

Construção de uma lista:
Em uma lista **simplesmente encadeada** e **não ordenada**, a inserção de um novo elemento funciona da seguinte forma:

1. Aloca-se memória para o novo nó.
2. O ponteiro `prox` do novo nó é configurado para apontar para o nó atual do cabeça.
3. O ponteiro `prox` do nó cabeça é atualizado para apontar para o novo nó, inserindo-o entre o nó cabeça e os outros nós da lista.

Busca em uma lista encadeada:
Para realizar uma busca:

1. **Lista não ordenada**: percorre-se todos os nós, comparando as chaves até encontrar uma correspondência ou atingir o fim da lista (quando o ponteiro `prox` é nulo).
2. **Lista ordenada**: a busca pode ser interrompida quando a chave do nó atual for maior que a chave buscada. No pior caso, é necessário verificar todos os nós.

Exemplo de código para busca em uma lista ordenada:

c

```
No* buscar(No* cabeça, int chave) {
```

```
    No* atual = cabeça->prox;
    while (atual != NULL && atual->chave < chave) {
        atual = atual->prox;
    }
    return (atual != NULL && atual->chave == chave) ? atual : NULL;
}
```

Esse código percorre a lista até encontrar a chave ou até atingir o fim da lista.

Código 5 - Busca em lista encadeada ordenada:
Este código realiza a busca em uma lista encadeada ordenada, retornando o endereço do nó encontrado ou `NULL` se o elemento não estiver presente.

c

```
No *buscar(No *no_cabeca, No **aux, int chave) {
    No *atual = no_cabeca->prox;
    *aux = no_cabeca;
    while (atual != NULL) {
        if (atual->chave < chave) {
            *aux = atual;
            atual = atual->prox;
        } else if (atual->chave == chave) {
            return atual; // Elemento encontrado
        } else {
            return NULL; // Elemento não encontrado
        }
    }
    return NULL; // Lista vazia
}
```

Observação:

- A função retorna o nó anterior ao elemento buscado (via o ponteiro `aux`), o que a torna útil também para inserção. Isso ocorre porque, ao contrário da alocação sequencial, a lista encadeada não permite acessar diretamente os elementos através de índices.
- Em uma **lista não ordenada**, a busca prossegue até o fim, sem necessidade de comparar a chave como no caso das listas ordenadas. A principal diferença ocorre apenas na inserção e remoção, que são basicamente as mesmas em ambas as listas.

A busca em listas encadeadas ordenadas exige a verificação da chave em cada nó, interrompendo a busca quando o elemento é encontrado ou se a chave do nó é maior que a buscada.

Código 6 - Inserção em lista encadeada:
A função insere um novo nó em uma lista encadeada, verificando se o nó a ser inserido já existe ou se a alocação falhou.

c

```
int inserir(No *no_ant, Elemento novo_elemento, int chave) {
    No *aux, *anterior = no_cabeca;
    No *novo_no = (No *) calloc(1, sizeof(No));
    aux = buscar(no_cabeca, &anterior, chave);
    if ((novo_no == NULL) || (aux != NULL))
        return 0; // Falha na inserção
    else {
```

```
        novo_no->elemento = novo_elemento;
        novo_no->chave = chave;
        novo_no->prox = anterior->prox;
        anterior->prox = novo_no;
        return 1; // Inserção bem-sucedida
    }
}
```

Explicação do processo de inserção (linhas 8 a 11):

1. **Linhas 8 e 9**: O novo nó recebe o valor do elemento e a chave.
2. **Linha 10**: O campo `prox` do novo nó aponta para o nó seguinte, que é o mesmo apontado por `anterior`.
3. **Linha 11**: O campo `prox` do nó anterior é atualizado para apontar para o novo nó, inserindo-o na lista entre `anterior` e o próximo nó.

Essa abordagem funciona para listas ordenadas ou não ordenadas.

Remoção de nó:
A remoção consiste em fazer o nó anterior apontar para o nó posterior ao que será removido e, em seguida, desalocar o nó removido com a função `free`.

Código 7 - Remoção em lista encadeada:
A função remove um nó de uma lista encadeada, atualizando o ponteiro do nó anterior e desalocando o nó removido.

```
c

remover(No *no_cabeca, int chave) {
    No *aux, *anterior = no_cabeca;
    aux = buscar(no_cabeca, &anterior, chave);
    if (aux != NULL) {
        anterior->prox = aux->prox;
        free(aux);
        return 1; // Remoção bem-sucedida
    } else
        return 0; // Falha na remoção
}
```

Explicação:

1. **Linha 3**: Busca o nó com a chave fornecida.
2. **Linha 5**: O ponteiro `prox` do nó anterior é atualizado para o próximo nó de `aux`.
3. **Linha 6**: O nó removido é desalocado com `free`.

A estrutura para listas duplamente encadeadas inclui um campo adicional `ant`, que aponta para o nó anterior, modificando a estrutura do nó conforme mostrado no **Código 8**.

Código 8 - Definição de nó para lista duplamente encadeada:

```
c

struct No {
    int chave;
    Elemento elemento;
    No *prox;
```

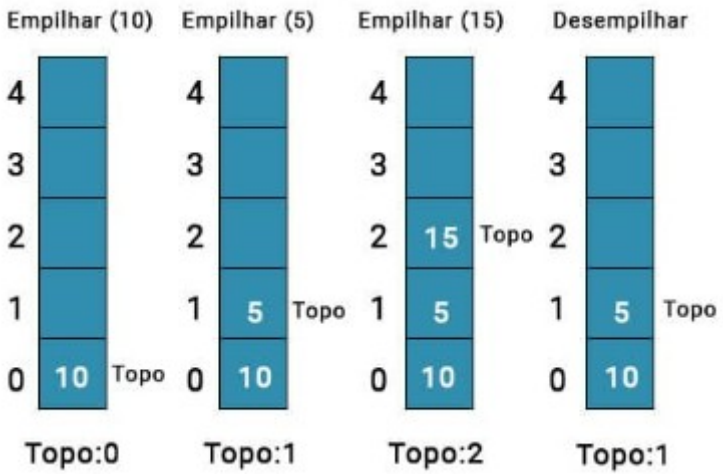
```
    No *ant;
}
```

Resumo:

- **Estrutura**: Um nó de lista duplamente encadeada contém um ponteiro `prox` (próximo nó) e um ponteiro `ant` (nó anterior).
- **Inserção e Remoção**: Requer ajustes devido à presença do ponteiro para o antecessor, tornando o processo mais complexo, mas a complexidade geral não é alterada.
- **Busca**: Funciona como em listas simplesmente encadeadas, mas pode ser simplificada.
- **Lista Circular**: Para criar uma lista circular, o último nó deve apontar para o nó cabeça. Em uma lista duplamente encadeada, o campo `ant` do nó cabeça também deve apontar para o último nó para permitir percorrê-la em ambos os sentidos.

Pilha - Tipo de Lista:

- **Definição**: A pilha é uma lista onde inserção, remoção e acesso ocorrem sempre na mesma extremidade, chamada de **topo**.
- **Propriedade**: Segue a regra **LIFO** (Last In, First Out), ou seja, o último elemento inserido é o primeiro a ser removido.
- **Importância**: Apenas a posição do topo precisa ser monitorada.
- **Caraterística**: Pilhas invertem a sequência dos elementos, pois a remoção acontece na ordem inversa da inserção.



Exemplo de Empilhamento e Desempilhamento:

- **Símbolos**:
  - { = base da pilha.
  - } = topo da pilha.
- **Elementos a inserir**: a, b, c, d, e (lidos da esquerda para a direita).
- **Tabela 1 – Empilhamento**:

Pilha	Operação	Sequência de Entrada
`{ }`	push(a)	a, b, c, d, e
`{ a }`	push(b)	b, c, d, e
`{ a, b }`	push(c)	c, d, e
`{ a, b, c }`	push(d)	d, e
`{ a, b, c, d }`	push(e)	e
`{ a, b, c, d, e }`		

- **Tabela 2 - Desempilhamento**:

Pilha	Operação	Sequência de Saída
`{ a, b, c, d, e }`	pop()	e
`{ a, b, c, d }`	pop()	e, d
`{ a, b, c }`	pop()	e, d, c
`{ a, b }`	pop()	e, d, c, b
`{ }`	pop()	e, d, c, b, a

Resumo: O empilhamento segue a ordem de inserção, enquanto o desempilhamento ocorre na ordem inversa, evidenciando a propriedade LIFO.

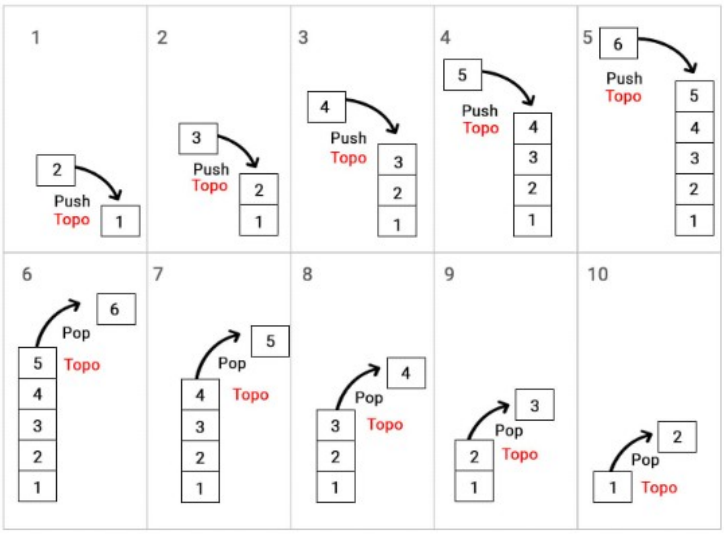
Resumo sobre Pilhas:

1. **Operações**:
    - **Push**: Insere um elemento na pilha, recebendo o item como parâmetro.
    - **Pop**: Remove o topo da pilha, sem parâmetros, atualizando a posição do topo.
2. **Sequência de entrada e saída**: A pilha segue a regra LIFO (Last In, First Out), ou seja, o último item inserido é o primeiro a ser removido. Na execução do exemplo, o topo foi sempre a extremidade direita.
3. **Execução das operações**: Embora as operações de empilhamento e desempilhamento sejam tipicamente sequenciais (empilhar todos os itens antes de desempilhar), nada impede que sejam mescladas. No entanto, tentar desempilhar de uma pilha vazia gera erro.
4. **Propriedade LIFO**: A pilha respeita a regra LIFO, onde o último elemento inserido é sempre o primeiro a ser removido. Isso impede desempilhar um item abaixo do topo, o que violaria a definição de pilha.
5. **Estrutura simplificada**: Pilhas podem ser implementadas de forma mais simples, sem a necessidade de um campo "chave", pois só se manipula o topo da pilha. Isso permite que elementos iguais sejam empilhados. Se usar alocação sequencial, o campo de ponteiro (para encadeamento) pode ser dispensado, com o nó limitando-se a armazenar o dado.

6. **Aplicações de pilhas**:
    - **Desfazer ações**: Em programas, ações são empilhadas para permitir o recurso de "desfazer", revertendo as ações na ordem inversa em que foram realizadas (do mais recente para o mais antigo).
    - **Execução de programas**: Durante a execução de um programa, quando uma função é chamada, o endereço de retorno é empilhado. Ao terminar, o programa retorna ao último endereço empilhado, permitindo o controle do fluxo de execução. Pilhas são essenciais para gerenciar chamadas de função e retornos.
7. **Representação e implementação**: Pilhas podem ser implementadas com alocação sequencial ou encadeada, com diferentes vantagens. A alocação sequencial tem o benefício de simplicidade, enquanto a encadeada permite uma maior flexibilidade de tamanho e estrutura.

Pilhas em Alocação Sequencial:

1. **Limitação de Memória**: Pilhas com alocação sequencial têm o limite de tamanho do vetor, exigindo cuidados para evitar *underflow* (desempilhamento de pilha vazia) e *overflow* (empilhamento em pilha cheia).
2. **Operações de Empilhamento e Desempilhamento**: Ambas ocorrem na mesma extremidade da pilha (topo). Não há necessidade de mover os outros elementos do vetor, pois as operações são feitas apenas no topo.
3. **Controle de Topo**: A posição do topo é controlada por uma variável do tipo inteiro. Quando a pilha está vazia, o topo é indicado por -1. A inserção de um elemento no topo incrementa o valor da variável *topo* e, no desempilhamento, o valor de *topo* é decrementado.
4. **Desalocação de Memória**: O desempilhamento não libera a memória de fato, mas apenas altera o índice do topo, deixando as posições a partir do topo até o limite do vetor disponíveis para novos elementos.
5. **Verificação de Espaço**: Antes de empilhar, verifica-se se há espaço disponível no vetor, comparando o valor de *topo* com o tamanho máximo do vetor. O empilhamento só ocorre se houver espaço.
6. **Falha no Empilhamento**: Se o vetor estiver cheio, tentar empilhar um elemento resultará em falha.



Código Completo:

1. **Função de Empilhamento (`push`)**:

```
c

int push(Elemento elemento) {
    if (topo < (MAX_PILHA - 1)) {  // Verifica se há espaço na pilha
        topo++;                    // Incrementa o topo
        pilha[topo] = elemento;    // Insere o elemento no topo
        return 1;                  // Retorna sucesso
    } else {
```

```
        return 0;                    // Retorna falha (pilha cheia)
    }
}
```

2. **Função de Desempilhamento (`pop`)**:

```
c

Elemento pop() {
    if (topo >= 0) {               // Verifica se a pilha não está vazia
        Elemento temp = pilha[topo]; // Armazena o elemento do topo
        topo--;                     // Decrementa o topo, liberando a posição
        return temp;                // Retorna o elemento desempilhado
    } else {
        // Retorna algum valor de erro ou um comportamento de falha
        return NULL;  // Exemplo de retorno quando a pilha está vazia
    }
}
```

Esses códigos implementam as operações de empilhamento e desempilhamento em uma pilha alocada sequencialmente, com o controle do topo e a verificação de limites para evitar overflow e underflow.

A **função de desempilhamento** (Código 10) é responsável por remover um elemento do topo da pilha em uma implementação de alocação sequencial. A função verifica se a pilha não está vazia, ou seja, se o valor de `topo` é maior ou igual a 0. Caso positivo, ela armazena o valor do topo em uma variável temporária (`valor_recuperado`), decrementa a variável `topo` e retorna o valor removido. Se a pilha estiver vazia (`topo < 0`), a função retorna `NULL`, indicando erro (underflow).

Código 10: Função de Desempilhamento:

```
c

Elemento pop (void) {
    Elemento valor_recuperado;
    if (topo >= 0) {
        valor_recuperado = pilha[topo];
        topo--;
        return valor_recuperado;
    } else {
        return NULL; //falha
    }
}
```

Exemplificação com Vetor de 3 Posições:

- Antes do primeiro **empilhamento**, o valor de `topo` é `-1`, indicando que a pilha está vazia.
- **`push(a)`**: O valor de `topo` é incrementado de -1 para 0, e o elemento "a" é inserido na posição 0 do vetor.
- **`push(b)` e `push(c)`**: Os valores de `topo` são incrementados para 1 e 2, e os elementos "b" e "c" são inseridos nas posições 1 e 2, respectivamente.
- **Tentativa de `push(d)`**: Nesse ponto, a pilha está cheia, já que o valor de `topo` é 2 e o valor máximo de `topo` seria 2 (MAX_PILHA - 1 = 2). Portanto, a tentativa de inserir "d" falha, pois a pilha não possui espaço disponível.

Após isso, **desempilhamentos** ocorrem:

- **Primeiro `pop()`**: O elemento "c" é removido, e o valor de `topo` é decrementado de 2 para 1.
- **Novo `push(d)`**: O valor de "d" é inserido, pois a posição 2 do vetor estava disponível após o desempilhamento de "c".
- **Seguindo com `pop()`**: O elemento "d" é removido, e o valor de `topo` é decrementado de 1 para 0.
- Com mais dois **`pop()`** subsequentes, os elementos "b" e "a" são removidos da pilha, e a pilha se torna vazia novamente.
- Ao tentar chamar **`pop()`** após a pilha estar vazia, o valor de `topo` será `-1`, o que impede a operação e resulta em falha (underflow).

Esse processo ilustra como as operações de empilhamento e desempilhamento funcionam e como o controle de `topo` é essencial para garantir que não ocorra o **overflow** (tentativa de empilhar em pilha cheia) ou **underflow** (tentativa de desempilhar em pilha vazia).

Pilhas em Alocação Dinâmica (Encadeada)

Em pilhas implementadas com alocação encadeada, não há limite fixo de tamanho, já que o empilhamento é limitado apenas pela memória disponível. O **desempilhamento** em uma pilha vazia deve ser prevenido, mas não há necessidade de verificar overflow, já que a alocação de memória é dinâmica e a pilha só "estoura" quando não há mais memória para alocar novos nós.

A pilha é implementada como uma lista **simplesmente encadeada**, sem a necessidade de um nó cabeça (inútil nesse caso). A **variável `topo`** é um ponteiro que aponta para o último nó inserido, e inicialmente tem o valor nulo em uma pilha vazia.

Operações:

- **Empilhamento (`push`)**: O processo envolve a alocação de um novo nó para armazenar o elemento. O campo `prox` desse nó aponta para o que era o topo da pilha, e o ponteiro `topo` é atualizado para apontar para esse novo nó.
- **Desempilhamento (`pop`)**: O ponteiro `topo` é ajustado para apontar para o nó anterior (campo `prox`), e o nó desempilhado é desalocado. Se a pilha estiver vazia, `topo` será nulo.

Exemplo de Código de Empilhamento (Código 11):

```
c

void push(Elemento elemento) {
    No* novo_no = (No*) malloc(sizeof(No)); // Aloca memória
    if (novo_no != NULL) {
        novo_no->elemento = elemento;    // Armazena o elemento
        novo_no->prox = topo;            // Novo nó aponta para o topo atual
        topo = novo_no;                  // Atualiza o topo para o novo nó
    } else {
        printf("Erro: Memória insuficiente.");
    }
}
```

A **alocação dinâmica** é feita na linha 2, e a linha 3 verifica se a alocação foi bem-sucedida. Se a alocação falhar, não é possível empilhar mais elementos. Se for bem-sucedida, o novo nó é inserido na pilha e o ponteiro `topo` é atualizado.

Em resumo, a pilha em alocação encadeada utiliza a alocação dinâmica de memória, onde cada elemento é armazenado em um nó encadeado, e as operações de empilhamento e desempilhamento manipulam os ponteiros para adicionar e remover elementos de forma eficiente.

Função de Empilhamento e Desempilhamento
A **função de empilhamento** (Código 11) implementa a operação `push` em uma pilha encadeada. Ela realiza os seguintes passos:

1. **Alocação dinâmica de memória** para o novo nó usando `calloc`.
2. Se a alocação for bem-sucedida (`novo_no != NULL`), o elemento é armazenado no nó, e o ponteiro `prox` deste nó aponta para o topo atual da pilha.
3. O ponteiro `topo` é atualizado para apontar para o novo nó.
4. A função retorna `1` para indicar sucesso e `0` para falha (caso a alocação não seja bem-sucedida).

Código de Empilhamento (Código 11):

c

```
int push(Elemento elemento) {
    No *novo_no = (No*) calloc(1, sizeof(No));  // Aloca memória para o novo nó
    if (novo_no != NULL) {
        novo_no->elemento = elemento;  // Armazena o elemento
        novo_no->prox = topo;          // Aponta para o topo atual
        topo = novo_no;                // Atualiza o topo
        return 1;                       // Sucesso
    } else {
        return 0;                       // Falha (memória insuficiente)
    }
}
```

Para o **desempilhamento**:

- A função de **desempilhamento** envolve a verificação se a pilha está vazia, impedindo o underflow.
- Se a pilha não estiver vazia, o ponteiro `topo` é atualizado para apontar para o próximo nó (linha 7), e a memória do nó desempilhado é liberada (linha 8).
- O elemento do nó desempilhado é armazenado em `elemento_recuperado` e retornado pela função.

Esse processo garante que os elementos sejam inseridos e removidos da pilha de maneira eficiente, manipulando dinamicamente os ponteiros para atualizar o topo.

Função de Desempilhamento (Código 12)
A **função de desempilhamento** (`pop`) remove um elemento do topo da pilha encadeada e realiza as seguintes etapas:

1. **Verifica se a pilha não está vazia** (`topo != NULL`).

2. **Armazena o elemento** do topo na variável `elemento_recuperado`.
3. **Atualiza o ponteiro `topo`** para apontar para o próximo nó.
4. **Libera a memória** do nó desempilhado usando `free(aux)`.
5. **Retorna o elemento** desempilhado.

Se a pilha estiver vazia, a função retorna `NULL` para indicar falha. O número de passos executados é fixo em ambas as operações.

Código de Desempilhamento (Código 12):

```
c

int pop(void) {
    No *aux;
    Elemento elemento_recuperado;
    if (topo != NULL) {
        elemento_recuperado = topo->elemento;  // Armazena o elemento do topo
        aux = topo;                            // Guarda o nó atual
        topo = topo->prox;                     // Atualiza topo para o próximo
nó
        free(aux);                             // Libera a memória do nó
desempilhado
        return elemento_recuperado;            // Retorna o elemento
desempilhado
    } else {
        return NULL;                           // Falha se a pilha estiver vazia
    }
}
```

Pilhas e Expressões Aritméticas Binárias

Expressões aritméticas envolvem **operadores**, que indicam operações aritméticas, **operandos**, que são os valores sobre os quais as operações atuam, e **delimitadores**, que definem a precedência das operações. A manipulação dessas expressões requer uma representação adequada em sistemas computacionais.

Expressões Aritméticas e Notação Infixa

A **notação infixa** é a forma tradicional de escrever expressões aritméticas, com operadores entre os operandos. No entanto, ela é **ambígua**, pois não define claramente a ordem de execução das operações, exigindo regras de precedência e delimitadores.

Exemplo: na expressão *2 * 3 / 4 * 2*, o resultado pode ser **3** ou **0,75**, dependendo da ordem de execução das operações. Para resolver isso, a **notação parentizada** é usada, onde os delimitadores (como parênteses) eliminam a ambiguidade. O desafio em avaliar essas expressões é que a prioridade das operações não segue a ordem de aparição.

Notação Polonesa e Polonesa Reversa

A **notação polonesa** coloca os operadores antes dos operandos, evitando ambiguidade. Por exemplo, a expressão infixa * (2 * 3) / (4 * 2) * é escrita como */ * 2 3 * 4 2* na notação polonesa. Isso elimina a necessidade de parentização e permite processar as operações na ordem em que aparecem.

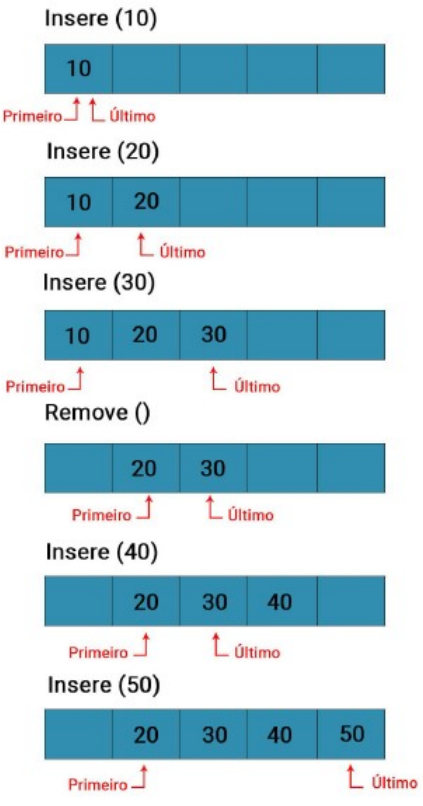
A **notação polonesa reversa** (ou pós-fixa) coloca os operadores após os operandos. Para converter expressões da notação infixa para a pós-fixa, usa-se pilhas, copiando os operandos diretamente e processando os operadores conforme sua prioridade.

Na notação pós-fixa, a expressão é percorrida, empilhando os operandos. Quando um operador é encontrado, dois operandos são desempilhados, a operação é realizada e o resultado é empilhado novamente. Ao final, o resultado final está no topo da pilha. Exemplo: para *2 3 * 4 2 * /*, a operação é realizada conforme a ordem estabelecida pela notação.

Execução de uma Expressão Pós-Fixa

1. **Passo 1**: O 3 é empilhado, depois o 2. Ao encontrar o operador de produto (*), 2 e 3 são desempilhados, multiplicados (2 * 3), e o resultado (6) é empilhado.
2. **Passo 2**: O 4 e o 2 são empilhados. Ao encontrar o operador de produto novamente, 4 e 2 são desempilhados, multiplicados (4 * 2), e o resultado (8) é empilhado.
3. **Passo 3**: Com a pilha contendo 6 e 8, ao encontrar o operador de divisão (/), 6 e 8 são desempilhados, divididos (8 / 6), e o resultado (0,75) é empilhado como resultado final da expressão.

Filas



- **Definição**: Uma fila é uma estrutura de dados em que as inserções ocorrem no final (retaguarda) e as remoções no início, seguindo o princípio **FIFO** (First In, First Out), ou seja, o primeiro a entrar é o primeiro a sair.
- **Controle**: Para implementar uma fila, são necessários dois controles: um para o início e outro para o fim da fila.
- **Operações**:
  - **Enfileirar (enqueue)**: Inserção de um elemento na fila.
  - **Desenfileirar (dequeue)**: Remoção do primeiro elemento da fila.
- **Exemplo**: Se a sequência de entrada for `a, b, c, d, e`, a operação de enfileirar coloca os elementos na fila, e a operação de desenfileirar os remove na mesma ordem:
  - Enfileirar: a → b → c → d → e.
  - Desenfileirar: a → b → c → d → e.
- **Problema**: Em filas com alocação sequencial, as remoções ocorrem pela esquerda, enquanto as inserções ocorrem pela direita, o que pode gerar dificuldades na implementação.
- **Aplicações**:
  - **Sistemas Operacionais**: Gestão de processos, como na execução de programas em CPUs com múltiplos núcleos.
  - **Impressoras**: Filas de impressão, onde os arquivos são processados na ordem de chegada.

- **Complexidade**: Filas têm diversas implementações, incluindo alocação sequencial e encadeada, que podem resolver problemas como a gestão de filas de múltiplos níveis ou com prioridades.

Filas em Alocação Sequencial

- **Problema**: Implementar filas em alocação sequencial exige o controle das posições inicial (remoções) e final (inserções) da fila, o que pode causar dois problemas:
    - A posição inicial se desloca conforme as remoções, criando espaços vazios no vetor.
    - Inserções podem ser bloqueadas quando o fim do vetor é alcançado, mesmo que existam espaços desocupados no início.
- **Deslocamento da fila**: Com inserções e remoções contínuas, a fila "desloca" sua posição, com índices sucessivamente maiores, o que gera desperdício de memória e bloqueio de novas inserções.
- **Soluções**:
    - **Deslocamento de elementos**: A cada remoção, os elementos são deslocados uma posição para frente, mantendo o início da fila sempre na primeira posição. No entanto, essa abordagem pode ser ineficiente.
    - **Lista Circular**: O vetor se comporta de forma circular, onde após o fim, a próxima posição é a primeira do vetor. Isso permite inserir elementos novamente após o fim do vetor, caso a primeira posição esteja livre.

Implementação Circular de Filas

- **Fila Circular**: Em uma fila circular, o vetor se comporta de forma que, ao alcançar o fim, o próximo índice é a primeira posição.
- **Controle das Posições**: Usam-se as variáveis “I” (início) e “F” (fim) para controlar a fila. Após uma inserção na N-ésima posição e ao menos uma remoção, F = N e I > 1.
- **Inserção Após Remoção**: Com uma remoção realizada, a primeira posição se torna disponível para novas inserções. Assim, após a remoção, o fim da fila passa a ser a primeira posição, resultando em F = 1 e I > 1.
- **Situação I > F**: A posição inicial (I) pode ser maior que a posição final (F), o que é uma característica da fila circular.

Fila Circular

- **Comportamento de Inserção e Remoção**:
    - **Inserção**: Ao inserir em uma fila circular, o final da fila (F) pode se mover até a última posição do vetor. Se houver remoções, a primeira posição (índice 1) poderá ser reutilizada, fazendo o fim da fila voltar à posição inicial.
    - **Remoção**: Quando o início da fila (I) chega à N-ésima posição, a remoção move I para a primeira posição do vetor, mantendo a condição de que $I \leq F$.
- **Fila com um Elemento**: Quando a fila contém apenas um elemento, I = F, e sua posição depende das operações feitas. Caso a fila esteja vazia, ambos I e F são -1.
- **Fila Vazia**: Uma fila vazia deve ser indicada por I = F = -1. Após a remoção do único elemento, ambos I e F devem ser -1, sinalizando que a fila está vazia.

- **Fila Cheia**:
    - Se não houver remoções, a fila estará cheia quando I = 1 e F = N.
    - Se houver remoções, a fila estará cheia quando I = F + 1, indicando que o início da fila está logo após o fim, completando o círculo.
- **Consideração de Linguagens como C**: Em linguagens como C, que começam a indexação em 0, a lógica da fila circular precisa ser ajustada, considerando o índice 0 como a posição inicial do vetor.

Esse comportamento é fundamental para a implementação de filas circulares em vetores, garantindo que a fila seja bem gerenciada, sem desperdício de memória e com as operações de inserção e remoção funcionando corretamente.

Função enfileirar (Código 13):

```
c

int enfileirar(Elemento elemento) {
    if (!( (inicio == 0 && fim == MAX_FILA - 1) || (inicio == fim + 1) )) {
        if ((fim == MAX_FILA - 1) || (fim == -1)) {
            fila[0] = elemento;
            fim = 0;
            if (inicio == -1)
                inicio = 0;
        } else {
            fila[++fim] = elemento;
            return 1; // sucesso
        }
    } else {
        return 0; // falha
    }
}
```

- **Função desenfileirar (Código 14)**:

```
c

Elemento desenfileirar(void) {
    Elemento elem_temp;
    if (inicio != -1) {
        elem_temp = fila[inicio];
        if (inicio == fim) {
            fim = inicio = -1;
        } else if (inicio == MAX_FILA - 1) {
            inicio = 0;
        } else {
            inicio++;
            return elem_temp;
        }
    } else {
        return NULL; // falha
    }
}
```

Comentário Geral:

- Em ambas as funções, o número de passos não varia, e os elementos não são desalocados diretamente. As posições dos elementos removidos ficam livres para novas inserções.

Filas em Alocação Dinâmica:

- **Implementação com Lista Encadeada**:
    - A fila é implementada por uma lista simplesmente encadeada, sem nó cabeça, utilizando dois ponteiros: `inicio` (para o primeiro elemento) e `fim` (para o último elemento). Inicialmente, ambos são `NULL`, indicando que a fila está vazia.
- **Inserção (`enfileirar`)**:
    - Quando um elemento é inserido, a memória é alocada para o nó e o endereço é armazenado. Se a fila estiver vazia, `inicio` e `fim` apontam para o mesmo nó.
    - Em inserções subsequentes, `inicio` permanece apontando para o primeiro nó, e `fim` é atualizado para o novo nó inserido, com o campo `prox` do nó anterior apontando para o novo nó.
- **Remoção (`desenfileirar`)**:
    - A remoção envolve mover o ponteiro `inicio` para o próximo nó na fila (`inicio = inicio->prox`). O nó removido é desalocado.
    - Se o único elemento for removido, ambos `inicio` e `fim` se tornam `NULL`, indicando que a fila está vazia.
- **Vantagens da Alocação Dinâmica**:
    - **Desperdício de Memória**: Ao contrário da alocação sequencial, a alocação dinâmica elimina o desperdício de memória, já que cada nó removido é desalocado e a memória pode ser reutilizada.
    - **Fila de Tamanho Dinâmico**: A fila pode crescer conforme necessário, limitada apenas pela memória disponível, sem o risco de estouro por limite de tamanho pré-definido como em alocação sequencial.
    - **Sem Implementação Circular**: Não há necessidade de implementação circular, pois a fila pode crescer conforme necessário. O campo `prox` do último nó sempre é `NULL`, o que simplifica a implementação.
- **Estrutura de Cada Nó**:
    - Cada nó possui:
        - `Elemento`: Contém o dado a ser armazenado.
        - `Prox`: Aponta para o próximo nó. O campo `prox` do último nó é `NULL`.
- **Funcionamento das Operações**:
    - **Enfileiramento**: Quando um novo nó é inserido, o campo `prox` do nó anterior é atualizado para apontar para o novo nó, e `fim` é atualizado para o novo nó inserido.
    - **Desenfileiramento**: A operação de remoção faz com que o ponteiro `inicio` avance para o próximo nó e o nó removido seja desalocado.

Esta abordagem oferece flexibilidade e eficiência, especialmente quando comparada à alocação sequencial.

Funções de Enfileiramento e Desenfileiramento, e Algoritmo de Ordenação por Filas:
Código 15: Função enfileirar

```
c
Copiar código
int enfileirar ( Elemento elemento ) {
    No *novo_no = ( No * ) calloc ( 1 , sizeof ( No ) );
    novo_no -> elemento = elemento;
    novo_no -> prox = NULL;
    if ( novo_no != NULL ) {
        if ( fim != NULL )
            fim->prox = novo_no;
        else {
            inicio = novo_no;
            fim = novo_no;
        }
        return 1; //sucesso
    } else
        return 0; //falha
}
```

- A função aloca memória para um novo nó com `calloc`, atribui o elemento e define o ponteiro `prox` como `NULL`.
- Se a fila não estiver vazia (`fim != NULL`), o ponteiro `fim` será atualizado para apontar para o novo nó. Caso contrário, `inicio` e `fim` são ambos atualizados para apontar para o novo nó (quando a fila está vazia).
- Se a alocação for bem-sucedida, retorna `1` (sucesso). Se falhar, retorna `0` (falha).

Código 16: Função desenfileirar

```
c
Copiar código
Elemento desenfileirar ( void ) {
    int elemento_recuperado;
    No *aux = inicio;
    if ( inicio != NULL ) {
        inicio = inicio->prox;
        if ( inicio == NULL )
            fim = NULL;
        elemento_recuperado = aux->elemento;
        free ( aux );
        return elemento_recuperado; //sucesso
    } else
        return NULL; //falha
}
```

- A função remove o elemento da frente da fila.
- Se a fila não estiver vazia, o ponteiro `inicio` avança para o próximo nó. Se a fila ficar vazia, `fim` também é ajustado para `NULL`.
- O nó removido é desalocado com `free` e o elemento é retornado.
- Se a fila estiver vazia, retorna `NULL` indicando falha.

Características Comuns:

- Ambas as funções têm complexidade constante, pois inserções e remoções são realizadas nas extremidades da fila, sem a necessidade de percorrer a lista.
- A alocação dinâmica de memória e o gerenciamento de ponteiros tornam a implementação eficiente, permitindo a reutilização de memória com a desalocação de nós.

Algoritmo de Ordenação por Filas:

- Utiliza **m filas** para ordenar **n** chaves inteiras, com base em uma base `m` (ex: base 10 para números decimais).
- **Passos**:
    1. Na primeira passagem, as chaves são distribuídas nas filas de acordo com o dígito menos significativo.
    2. As filas são concatenadas em uma nova lista.
    3. Na segunda passagem, o processo é repetido para o próximo dígito mais significativo, e assim por diante até a ordenação ser concluída.

Esse tipo de ordenação é conhecido como **ordenação por distribuição** e é especialmente útil para ordenar números inteiros.

Essas abordagens ilustram como a estrutura de filas, seja com alocação encadeada ou sequencial, pode ser aplicada tanto para operações simples como enfileiramento/desenfileiramento quanto em algoritmos de ordenação mais complexos.

Algoritmo 5: Ordenação por Distribuição
O algoritmo de ordenação por distribuição organiza uma lista de inteiros com base nos seus dígitos, utilizando múltiplas filas para agrupar os elementos conforme cada dígito significativo.

Passos principais:

1. **Entrada e Preparação:**
    - A função recebe um valor `m` (base de ordenação) e `n` (número de elementos a ordenar).
    - Define-se a quantidade de filas necessárias como `m-1` e prepara-se a variável auxiliar `aux`.
2. **Distribuição dos Elementos:**
    - Para cada elemento em `Entrada[]`, o dígito menos significativo é calculado e usado para indexar a fila correspondente.
    - Cada elemento é colocado na fila de acordo com o dígito analisado.
3. **Reorganização dos Elementos:**
    - As filas são processadas, e os elementos são recolocados na lista de entrada (`Entrada[]`), mantendo a ordem de distribuição.

Código Completo:

```
c
Copiar código
ordenacaoPorDistribuicao(int m, int n, int Entrada[]) {
    int nr_digito = m - 1, aux;
    int Fila[nr_digito];

    for (int i = 0; i < m - 1; i++) {
        for (int j = 0; j < n; j++) {
            aux = o i-ésimo dígito menos significativo de Entrada[j];
            Fila[aux] = Entrada[j];
        }
    }

    for (aux = 0; aux < m - 1; aux++) {
```

```
        while (Fila[aux] tiver elemento não processado) {
            Entrada[j] = Fila[aux];
            j++;
        }
    }
}
```

Objetivo: O algoritmo distribui os elementos entre as filas com base nos seus dígitos e, em seguida, reconstrói a lista a partir das filas, ordenando os elementos.

Ordenação
A ordenação é um problema central em computação, utilizado para resolver problemas mais complexos. O objetivo é encontrar uma permutação de uma sequência de dados que atenda a uma relação de ordem definida, como "menor que" para ordem crescente ou "maior que" para ordem decrescente.

Enunciado informal:
Dada uma sequência de dados, a tarefa é organizar os elementos de acordo com uma relação de ordem, como "menor que" para ordenar em ordem crescente.

Exemplo:
Para a sequência 12, 35, 17, 92, 45, 8, a solução em ordem crescente seria 8, 12, 17, 35, 45, 92 e em ordem decrescente seria 92, 45, 35, 17, 12, 8.

Formalização:
Seja S = s1, s2, ..., sn uma sequência de números inteiros distintos. Ordenar a sequência significa encontrar uma permutação S' = s'1, s'2, ..., s'n tal que para todo $0 < i < n$, $s'i < s'i+1$.

Embora o exemplo use inteiros, objetos de dados diferentes, como cadeias de caracteres, também podem ser ordenados, desde que se defina uma relação de ordem, como a ordem lexicográfica para strings.

Algoritmo de ordenação:
Um algoritmo de ordenação recebe uma sequência de dados e gera uma permutação que satisfaz a relação de ordem definida.

Classificação dos Algoritmos de Ordenação
Os algoritmos de ordenação podem ser classificados com base em diversos critérios:

1. **Complexidade Computacional**
2. **Complexidade de Espaço**
3. **Ordenação Interna ou Externa**
4. **Caráter Recursivo**
5. **Estabilidade**

Complexidade Computacional

A complexidade computacional estuda o número de operações necessárias para resolver um problema. A análise de complexidade utiliza a **notação O**, que define a cota assintótica superior do número de operações em função do tamanho da entrada (n). A notação indica o desempenho do algoritmo conforme o tamanho da instância cresce.

Um algoritmo é dito ser **O(f)** se, para um n suficientemente grande, a execução do algoritmo não exceder k * f(n), onde k é uma constante positiva. A eficiência de um algoritmo depende dessa análise, classificando-o em diferentes ordens de execução. Exemplos incluem:

- **O(1)**: Execução constante.
- **O(log n)**: Crescimento logarítmico.
- **O(n)**: Crescimento linear.
- **O(n log n)**: Crescimento linear-logarítmico.
- **O(n^2)**: Crescimento quadrático.

Essas ordens determinam o desempenho do algoritmo à medida que o tamanho da entrada aumenta.

Complexidade de Algoritmos de Busca e Ordenação
A busca binária e a busca linear exemplificam a diferença de complexidade computacional entre algoritmos. A busca binária, que ocorre em um vetor ordenado, compara a chave buscada com o elemento central e continua a busca na metade superior ou inferior do vetor, dependendo do caso. Sua complexidade é **O(log n)**, enquanto a busca linear, que examina cada elemento do vetor, tem complexidade **O(n)**.

Exemplo de busca binária: Para um vetor de 10^6 elementos, a busca linear exigiria 10^6 comparações, enquanto a busca binária apenas $\mathbf{\log_2(10^6) \approx 20}$ comparações, mostrando uma grande diferença na quantidade de operações.

Busca Binária - Código:

```
c
Copiar código
int buscaBinaria(int vetor[], int chave, int n) {
    int inicio = 0, fim = n - 1, meio;
    while (inicio <= fim) {
        meio = (inicio + fim) / 2;
        if (vetor[meio] == chave)
            return meio; // Encontro da chave
        else if (vetor[meio] < chave)
            inicio = meio + 1; // Busca na metade superior
        else
            fim = meio - 1; // Busca na metade inferior
    }
    return -1; // Chave não encontrada
}
```

Limite Inferior de Ordenação: Algoritmos de ordenação baseados em comparações entre elementos têm um limite inferior de complexidade de **O(n log n)**. Esse limite é demonstrado ao modelar a ordenação como um problema de decisão, representado por uma árvore de decisão binária. A altura mínima dessa árvore é **O(log n!)**, o que leva à conclusão de que a complexidade mínima de comparações para ordenar é **O(n log n)**.

Ordenação por Comparação - Modelo de Árvore de Decisão:

- Cada nó interno representa uma comparação entre dois elementos.
- As folhas representam as permutações dos elementos.
- A árvore é binária, com dois resultados possíveis para cada comparação: **x < y** ou **x > y**.

- A altura mínima da árvore de decisão é proporcional a **log n!**, levando ao limite inferior **O(n log n)**.

Complexidade de Espaço: Mede a memória necessária para a execução de um algoritmo. Algoritmos como o de contagem (counting sort), que usa um vetor auxiliar para contar a ocorrência de valores dentro de um intervalo específico, têm complexidade de tempo **O(n)**, mas podem ter alta complexidade de espaço. Nesse caso, a memória necessária é proporcional ao tamanho do conjunto universo das chaves, e não ao tamanho da entrada, o que caracteriza esses algoritmos como **pseudolineares**. Isso significa que, apesar de eficientes em termos de tempo, esses algoritmos podem ser ineficientes em termos de espaço.

Algoritmo de Contagem (Counting Sort):

```
c
Copiar código
void countingSort(int V[], int n) {
    int aux[100000] = {0}; // Vetor auxiliar
    for (int i = 0; i < n; i++) {
        aux[V[i]] = 1; // Marcar presença no intervalo [0, 100000]
    }
    int j = 0;
    for (int i = 0; i < 100000; i++) {
        if (aux[i] == 1) {
            V[j++] = i; // Coloca os valores ordenados de volta em V
        }
    }
}
```

Este comportamento destaca a importância de considerar tanto a complexidade de tempo quanto de espaço ao analisar a eficiência de um algoritmo. Algoritmos como o Counting Sort são rápidos em termos de tempo, mas podem ser ineficientes quando o conjunto universo das chaves é muito grande, como mostrado no exemplo acima.

Ordenação Interna x Externa

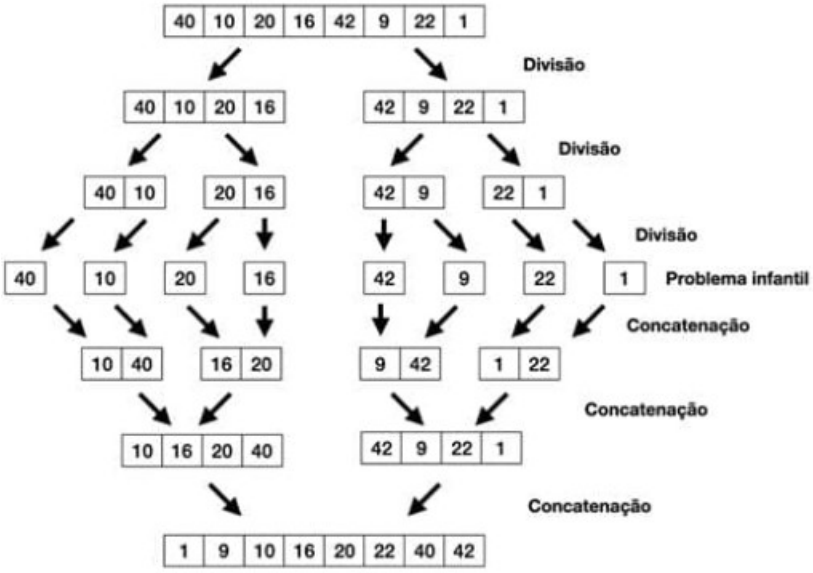
A ordenação interna e externa se diferenciam pelo tipo de memória onde os dados são armazenados. A memória principal (RAM) é de acesso aleatório e muito mais rápida, com tempo de acesso na ordem de nanosegundos ($1x10^{-9}$). Já a memória secundária, como o disco rígido ou SSD, tem acesso por blocos (geralmente de 512 Kbytes) e é mais lenta, com tempo de acesso na ordem de milissegundos ($1x10^{-3}$).

- **Ordenação Interna**: Os algoritmos operam com todos os dados na memória principal.
- **Ordenação Externa**: Os algoritmos lidam com dados armazenados na memória secundária. Contudo, devido ao paradigma de von Neumann, mesmo esses algoritmos precisam transferir parte dos dados para a memória principal para processamento. Após o processamento, o resultado é gravado novamente na memória secundária.

A principal distinção está na capacidade de operar com dados que não cabem inteiramente na memória principal, característica dos algoritmos de ordenação externa.

Caráter Recursivo nos Algoritmos de Ordenação

Alguns algoritmos de ordenação são **recursivos**, usando a técnica **dividir para conquistar**, enquanto outros são sequenciais. O **Merge Sort** é um exemplo de algoritmo recursivo.



Funcionamento do Merge Sort:

1. **Divisão do vetor**: O vetor é repetidamente dividido em duas metades até que as sublistas contenham apenas um elemento, considerado ordenado.

   Exemplo:

   - Vetor inicial: [8, 3, 5, 2, 1, 7, 4, 6]
   - Divide-se em: [8, 3, 5, 2] e [1, 7, 4, 6]
   - Divide novamente: [8, 3] e [5, 2], depois [1, 7] e [4, 6], e assim por diante até as sublistas terem um elemento.
2. **Concatenação**: Após a divisão, começa-se a **concatenar** os elementos ordenados:

   - Comparando dois elementos em cada divisão, coloca-se o menor na primeira posição do novo vetor. Este processo é repetido até o vetor estar completamente ordenado.

   Exemplo de concatenação:

   - Para unir [3, 8] e [2, 5], compara-se o primeiro de cada vetor e coloca-se o menor primeiro: [2, 3, 5, 8].

Características Recursivas:

- A cada nível de recursão, o vetor é dividido ao meio, o que cria uma árvore binária de decisões.
- A profundidade dessa árvore de recursão é $\log_2 n$, onde **n** é o número de elementos no vetor.
- Cada nível de concatenação exige **n** comparações, resultando em uma **complexidade de tempo O(n log n)**.

Complexidade do Merge Sort:

- O Merge Sort tem **complexidade O(n log n)**, pois a divisão do vetor acontece em $\log_2 n$ níveis, e em cada nível são feitas **n** comparações.
- Esse desempenho é eficiente, já que o algoritmo realiza a ordenação em tempo proporcional a **n log n** comparações.

A abordagem recursiva do Merge Sort envolve repetidas divisões e concatenações, tornando-o um exemplo típico de **dividir para conquistar** com **complexidade O(n log n)**.

Estabilidade nos Algoritmos de Ordenação

Um **algoritmo de ordenação é estável** quando mantém a ordem relativa dos elementos que já estão corretamente ordenados. Ou seja, se uma sequência já ordenada for passada para um algoritmo estável, nenhuma troca será feita.

A **estabilidade** pode reduzir o número de operações em **melhores casos**, mas não influencia a complexidade computacional teórica, especialmente em **piores casos**. Portanto, a estabilidade não afeta a análise assintótica do algoritmo.

Algoritmos Não Baseados em Comparação

Algoritmos de ordenação típicos comparam elementos da sequência para ordená-los, mas **algoritmos não baseados em comparação** não realizam essas comparações diretas. Para esses, o limite inferior de O(n log n) não se aplica, pois a premissa da árvore de decisão não é válida. Assim, é possível que esses algoritmos tenham **complexidade inferior a O(n log n)**.

Um exemplo é o **bucket sort** (método do balde), que ordena inteiros com base no número máximo de dígitos conhecidos. Ele realiza iterações por ordem numérica, separando os elementos por dígito e, ao final, concatena a sequência ordenada.

Exemplo de Bucket Sort

Dado a sequência de números de três dígitos: 005, 235, 014, 236, 423, 456, 890, o **Bucket Sort** é realizado em três etapas:

1. **Etapa 1**: Separação por centena:
    - Centena 0: 005, 014
    - Centena 2: 235, 236
    - Centena 4: 423, 456
    - Centena 8: 890
2. **Etapa 2**: Separação por dezena:
    - Centena 0: Dezena 0: 005, Dezena 1: 014
    - Centena 2: Dezena 3: 235, 236
    - Centena 4: Dezena 2: 423, Dezena 5: 456
    - Centena 8: Dezena 9: 890
3. **Etapa 3**: Separação por unidade:
    - Centena 0: Dezena 0: Unidade 5: 005, Dezena 1: Unidade 4: 014
    - Centena 2: Dezena 3: Unidade 5: 235, Unidade 6: 236
    - Centena 4: Dezena 2: Unidade 3: 423, Dezena 5: Unidade 6: 456
    - Centena 8: Dezena 9: Unidade 0: 890

Concatenando a sequência ordenada: 005, 014, 235, 236, 423, 456, 890.

A complexidade desse algoritmo é discutida. Alguns autores sugerem **O(n)**, considerando que o número de dígitos é constante e cada iteração percorre a sequência uma vez. Outros argumentam que a complexidade é **O(n log n)**, devido ao fato de k, o número de dígitos, ser log(n).

Método da Bolha (Bubble Sort)

O **Método da Bolha** verifica se uma sequência de números inteiros está ordenada comparando elementos adjacentes (si e si+1). Se para todos os pares si < si+1, a sequência está ordenada.

Exemplo: Para a sequência S = [10, 12, 15, 20, 25], as comparações mostram que si < si+1 para todos os elementos, indicando que a sequência está ordenada.

O método de ordenação trabalha trocando elementos adjacentes sempre que a condição si < si+1 não for atendida, explorando a transitividade das comparações (se si < si+1 e si+1 < si+2, então si < si+2).

Método da Bolha (Bubble Sort)
O **Método da Bolha** ordena uma sequência de números inteiros comparando elementos adjacentes. Se si < si+1, nada é feito; se não, eles são trocados. Esse processo é repetido até que nenhuma troca seja realizada em uma iteração, o que indica que a sequência está ordenada.

Exemplo: Para a sequência S = [15, 20, 8, 16, 40]:

- **1ª iteração**:
    - Passo 1: 15 < 20 (sem troca)
    - Passo 2: 20 > 8 (troca -> [15, 8, 20, 16, 40])
    - Passo 3: 20 > 16 (troca -> [15, 8, 16, 20, 40])
    - Passo 4: 20 < 40 (sem troca)

  Após a 1ª iteração, a sequência é [15, 8, 16, 20, 40].
- **2ª iteração**:
    - Passo 1: 15 > 8 (troca -> [8, 15, 16, 20, 40])
    - Passo 2: 15 < 16 (sem troca)
    - Passo 3: 15 < 20 (sem troca)
    - Passo 4: 15 < 40 (sem troca)

  Após a 2ª iteração, a sequência é [8, 15, 16, 20, 40].
- **3ª iteração**:
    - Passo 1: 8 < 15 (sem troca)
    - Passo 2: 15 < 16 (sem troca)
    - Passo 3: 16 < 20 (sem troca)
    - Passo 4: 20 < 40 (sem troca)

  Como não houve trocas, a sequência está ordenada após a 3ª iteração: [8, 15, 16, 20, 40].

O algoritmo finaliza quando uma iteração não realiza nenhuma troca, garantindo que a sequência está ordenada.

Complexidade e Características do Método da Bolha
A complexidade do algoritmo **Método da Bolha** é analisada considerando o **pior caso**. Quando a sequência é fornecida em ordem reversa, o algoritmo executa **$(n-1)^2$ comparações**, o que resulta em **$O(n^2)$** no pior cenário. Cada iteração posiciona um número em sua posição correta, com o maior valor sendo colocado na última posição a cada iteração.

Características:

- **Pior caso**: Ordem reversa, resultando em $O(n^2)$ comparações.
- **Melhor caso**: Sequência já ordenada, com $O(n-1)$ comparações.
- **Estabilidade**: O algoritmo não troca elementos que já estão corretamente ordenados.
- **Não recursivo**: Executa de forma iterativa.
- **Complexidade de espaço**: $O(n)$, usando memória apenas para armazenar a sequência.

Código em C:

```
c
Copiar código
void bolha(int *v) {
    int troca = 1;
    int i = 0;
    int aux;
    while (troca) {
        troca = 0;
        while (i < TAMANHO - 1) {
            if (v[i] > v[i+1]) {
                aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
                troca = 1;
            }
            i++;
        }
        i = 0;
    }
}
```

O algoritmo é eficiente em termos de espaço e fácil de implementar, mas não é adequado para sequências grandes devido à sua complexidade quadrática.

Método de Seleção (Selection Sort)
O **Método de Seleção** é um algoritmo de ordenação simples e iterativo. Ele trabalha selecionando o menor elemento da sequência a cada iteração e trocando-o com a posição correta.

Funcionamento:

1. Na primeira iteração, o menor elemento é encontrado e trocado com o primeiro.
2. Na segunda iteração, o menor elemento entre os elementos restantes é trocado com o segundo, e assim por diante até que a sequência esteja ordenada.

Exemplo (sequência: 13, 25, 8, 19, 7, 52):

- 1ª iteração: 7 é o menor.
- 2ª iteração: 8 é o menor.
- 3ª iteração: 13 é o menor.
- E assim sucessivamente até a sequência ordenada: 7, 8, 13, 19, 25, 52.

Complexidade Computacional:

- O número de comparações é: $(n-1) + (n-2) + ... + 1 = O(n^2)$.
- O algoritmo executa $O(n^2)$ em qualquer caso, não há diferença entre o pior, melhor ou caso médio.

Características:

- **Estabilidade**: Pode ser estável dependendo da implementação, mas geralmente não é.
- **Não recursivo**: Iterativo.
- **Complexidade de espaço**: O(n).

Código em C (Versão menos estável):

```
c
Copiar código
void selecao(int *v) {
    int i, j, aux;
    for (i = 0; i < TAMANHO - 1; i++) {
        for (j = i + 1; j < TAMANHO; j++) {
            if (v[i] > v[j]) {
                aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
        }
    }
}
```

Melhorando a Estabilidade (Versão estável):

```
c
Copiar código
void selecao(int *v) {
    int i, j, aux, minimo, pos_minimo;
    for (i = 0; i < TAMANHO - 1; i++) {
        minimo = v[i];
        pos_minimo = i;
        for (j = i + 1; j < TAMANHO; j++) {
            if (minimo > v[j]) {
                minimo = v[j];
                pos_minimo = j;
            }
        }
        if (pos_minimo != i) {
            aux = v[pos_minimo];
            v[pos_minimo] = v[i];
            v[i] = aux;
        }
    }
}
```

O método é eficiente em termos de espaço, mas sua complexidade quadrática torna-o ineficiente para grandes conjuntos de dados.

Método da Seleção (Selection Sort)
O **Método de Seleção** é um algoritmo de ordenação simples e iterativo. Ele funciona em etapas onde a cada iteração o menor elemento da sequência não ordenada é selecionado e trocado com o primeiro elemento da sequência não ordenada.

Funcionamento:

1. **Primeira iteração**: O algoritmo encontra o menor elemento da sequência e o troca com o primeiro. Após essa iteração, o menor elemento estará na posição correta (início da sequência ordenada).
2. **Segunda iteração**: O algoritmo encontra o menor elemento entre os elementos restantes e o troca com o segundo, colocando-o na segunda posição correta da sequência ordenada.
3. Esse processo continua até que a sequência esteja completamente ordenada. O número de iterações é **n-1**, onde **n** é o número de elementos.

Exemplo:

Sequência inicial: **13, 25, 8, 19, 7, 52**

- **1ª iteração**: O menor é 7, sequência: **7, 25, 8, 19, 13, 52**
- **2ª iteração**: O menor é 8, sequência: **7, 8, 25, 19, 13, 52**
- **3ª iteração**: O menor é 13, sequência: **7, 8, 13, 19, 25, 52**
- **4ª iteração**: O menor é 19, sequência: **7, 8, 13, 19, 25, 52**
- **5ª iteração**: O menor é 25, sequência: **7, 8, 13, 19, 25, 52**
- **6ª iteração**: O menor é 52, sequência final: **7, 8, 13, 19, 25, 52**

Complexidade Computacional:

- O algoritmo realiza **n-1** comparações na primeira iteração, **n-2** na segunda, e assim sucessivamente até **1** comparação na última iteração.
- O total de comparações é a soma de **(n-1) + (n-2) + ... + 1**, que resulta em **$O(n^2)$**.
- A complexidade é a mesma para todos os casos (melhor, pior e caso médio), já que o número de comparações não depende da entrada, mas apenas do número de elementos.

Características:

- **Estabilidade**: Pode ser instável dependendo da implementação, pois troca elementos sem se preocupar com a ordem de elementos iguais.
- **Não recursivo**: Algoritmo iterativo.
- **Complexidade de espaço**: **O(n)**, pois o algoritmo só precisa de espaço extra para armazenar a sequência a ser ordenada.

Código em C:

A versão **não estável** do algoritmo:

```
c
Copiar código
void selecao(int *v) {
    int i, j, aux;
    for (i = 0; i < TAMANHO - 1; i++) {
        for (j = i + 1; j < TAMANHO; j++) {
            if (v[i] > v[j]) {
                aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
        }
    }
```

```
}
```

Para tornar o algoritmo **estável**, realizamos duas etapas:

1. **Encontra-se o menor elemento**.
2. **Troca-se o menor elemento** com o elemento da posição correta.

```
c
Copiar código
void selecao(int *v) {
    int i, j, aux, minimo, pos_minimo;
    for (i = 0; i < TAMANHO - 1; i++) {
        minimo = v[i];
        pos_minimo = i;
        for (j = i + 1; j < TAMANHO; j++) {
            if (minimo > v[j]) {
                minimo = v[j];
                pos_minimo = j;
            }
        }
        if (pos_minimo != i) {
            aux = v[pos_minimo];
            v[pos_minimo] = v[i];
            v[i] = aux;
        }
    }
}
```

Conclusão:

O algoritmo **Selection Sort** é fácil de entender e implementar, mas sua **complexidade $O(n^2)$** o torna ineficiente para grandes conjuntos de dados. Ele é estável se implementado corretamente, mas não é a escolha ideal quando se busca eficiência para grandes volumes de dados.

Método da Inserção (Insertion Sort)
O **Método da Inserção** é um algoritmo eficiente para listas quase ordenadas. Ele funciona dividindo a lista em duas partes: uma ordenada e uma não ordenada. Inicialmente, a sequência ordenada contém apenas o primeiro elemento. O algoritmo insere o primeiro elemento da parte não ordenada na posição correta da parte ordenada. Este processo é repetido até que todos os elementos estejam ordenados.

Funcionamento:

1. **Primeira iteração**: O segundo elemento é comparado com o primeiro e, se necessário, é inserido na posição correta.
2. **Segunda iteração**: O terceiro elemento é comparado com os dois primeiros e é colocado na posição correta.
3. O processo continua até que todos os elementos sejam inseridos na parte ordenada da lista.

Exemplo (sequência inicial: 16, 8, 20, 18, 9, 2):

- **1ª iteração**: 8, 16, 20, 18, 9, 2
- **2ª iteração**: 8, 16, 20, 18, 9, 2
- **3ª iteração**: 8, 16, 18, 20, 9, 2

- **4ª iteração**: 8, 9, 16, 18, 20, 2
- **5ª iteração**: 2, 8, 9, 16, 18, 20

Complexidade:

- A **complexidade computacional** é **O(n²)**, já que no pior caso (sequência inversa), o algoritmo realiza um número crescente de comparações e trocas a cada iteração.
- O número total de comparações e trocas é dado pela soma de **1 + 2 + 3 + ... + n-1**, o que resulta em **O(n²)**.
- **Pior caso**: Sequência ordenada em ordem reversa.

Características:

- O algoritmo é **estável** (não altera a ordem de elementos iguais).
- **Não recursivo**.
- **Complexidade de espaço** é **O(n)**, pois o algoritmo usa espaço adicional para armazenar a sequência a ser ordenada.

Código em C:

```
c
Copiar código
void insertion(int *v) {
    int i, j, aux;
    for (i = 0; i < TAMANHO - 1; i++) {
        j = i + 1;
        while (v[j - 1] > v[j] && j > 0) {
            aux = v[j - 1];
            v[j - 1] = v[j];
            v[j] = aux;
            j--;
        }
    }
}
```

Conclusão:

O **Insertion Sort** é eficiente para listas quase ordenadas, mas sua complexidade de **O(n²)** torna-o ineficaz para grandes conjuntos de dados. É estável e simples, sendo ideal para casos em que os dados já estão parcialmente ordenados.

Introdução a Árvores

As árvores são estruturas de dados fundamentais na computação, amplamente utilizadas em diversas aplicações, como organização de diretórios, indexação em bancos de dados e interfaces gráficas.

Definição:

- **Estrutura hierárquica:** Os elementos estão relacionados por uma relação de ancestralidade.
- **Definição recursiva:** A estrutura mantém sempre válida a definição de árvore.

Objetivos:

1. **Conceitos básicos e propriedades**: Apresentar os fundamentos de árvores.
2. **Árvore binária de busca**: Estudar sua aplicação, relevância e características.
3. **Tipos de árvores de pesquisa**: Analisar diferentes tipos e os problemas enfrentados.
4. **Atividades práticas**: Consolidar os conceitos e habilidades adquiridos.

As árvores oferecem uma base importante para resolução de problemas e são uma ferramenta essencial na computação.

Introdução às Árvores em Computação
As árvores são estruturas de dados que otimizam operações como inserção, remoção, atualização e seleção de valores mínimo/máximo, sendo amplamente utilizadas devido à sua eficiência.

Objetivos desta seção:

1. **Compreender conceitos e propriedades básicas**: Introdução à estrutura e terminologia essencial.
2. **Conexão entre definições**: Gradualmente, construir uma visão integrada das árvores.
3. **Preparação para estudos avançados**: Estabelecer fundamentos para discussões mais elaboradas em módulos posteriores.

Os conceitos e propriedades serão destacados para facilitar o estudo e referência futura.

Conceitos Básicos e Terminologia de Árvores

Definição de Árvore

- Uma **árvore** é uma estrutura hierárquica composta por um conjunto finito de elementos chamados **nós** ou **vértices**, organizados segundo as seguintes regras:
    1. **Árvore vazia**: Conjunto sem elementos.
    2. **Árvore não vazia**:
        - **Nó raiz**: Nó único no topo da hierarquia.
        - **Subárvores**: Divisões disjuntas e não vazias do nó raiz, cada uma sendo uma nova árvore.
- **Propriedade Recursiva**: Cada subárvore é uma árvore completa, com seu próprio nó raiz e estrutura independente.

Características Fundamentais

1. **Hierarquia**: A organização reflete relações ancestrais entre os nós, com um nó raiz no topo e subárvores organizadas abaixo.
2. **Ausência de Ciclos**: Nenhum nó pode pertencer a mais de uma subárvore ao mesmo tempo, garantindo que a árvore seja acíclica.
3. **Floresta**: Um conjunto de árvores independentes forma uma floresta.

Explicação Detalhada

- Árvores podem ser definidas recursivamente: o conjunto de nós é vazio (árvore vazia) ou contém um nó raiz com subárvores disjuntas. A ausência de ciclos significa que nenhum nó participa de duas subárvores ao mesmo tempo, evitando relações circulares.
- Exemplo prático de diferença:
    - **Árvore (Figura 1)**: Estrutura acíclica com hierarquia bem definida.
    - **Grafo (Figura 2)**: Possui ciclos, como no exemplo onde o nó "D" pertence a dois conjuntos simultaneamente.

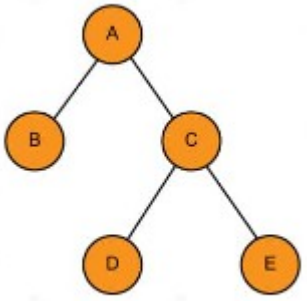


Figura 1: Árvore.

Representações Gráficas

1. **Clássica Hierárquica (Figura 1)**:
    - A raiz está no topo, com os nós conectados abaixo, evidenciando a hierarquia.
2. **Hierárquica Invertida (Figura 3)**:
    - A raiz está na base, com as subárvores ascendendo.
3. **Diagramas Alternativos**:
    - **Diagramas de Inclusão (Figura 4)**: Representam subárvores como conjuntos aninhados.
    - **Diagramas de Barras (Figura 5)**: Mostram a estrutura como barras horizontais ou verticais.

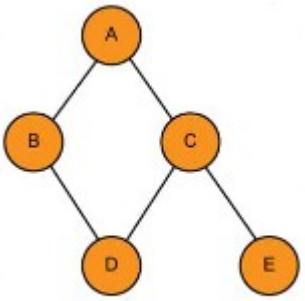


Figura 2: Grafo.

Destaques e Observações

- A **forma hierárquica clássica (Figura 1)** é a mais utilizada em Computação devido à clareza das relações hierárquicas.
- Outras representações são úteis para situações específicas, mas mantêm a essência da hierarquia dos nós.

Esse conhecimento é essencial para entender propriedades e operações em árvores, que serão explorados nas próximas seções.

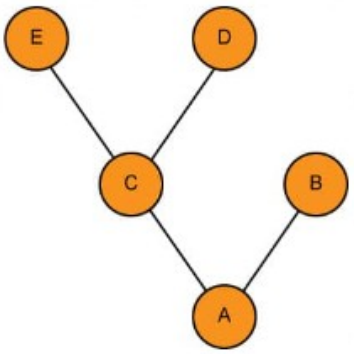


Figura 3: Representação hierárquica invertida.

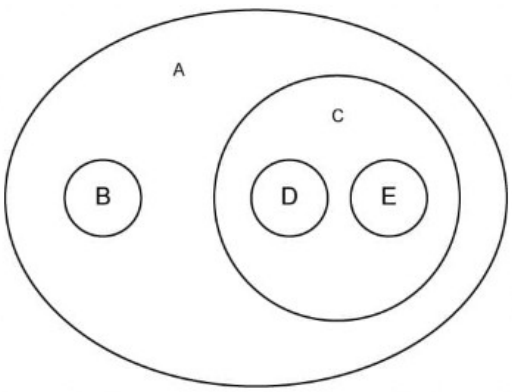


Figura 4: Representação por diagramas de inclusão.

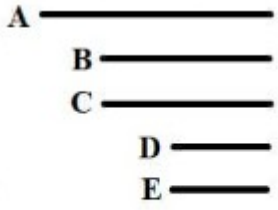


Figura 5: Representação por diagramas de barras.

Resumo Detalhado - Forma Parentizada e Relações Hierárquicas em Árvores

Forma Parentizada

- Representa hierarquias em árvores utilizando **parênteses aninhados.**
  - Exemplo: A árvore da Figura 1 é representada como `(A (B) (C (D) (E)))`.
- Semelhança com expressões aritméticas:
  - **Árvores** podem representar expressões matemáticas completamente parentizadas.
  - Operadores são posicionados nos nós, enquanto operandos ocupam subárvores.
  - Exemplo: A expressão `(a * (b + c / d))` equivale à árvore da Figura 6.

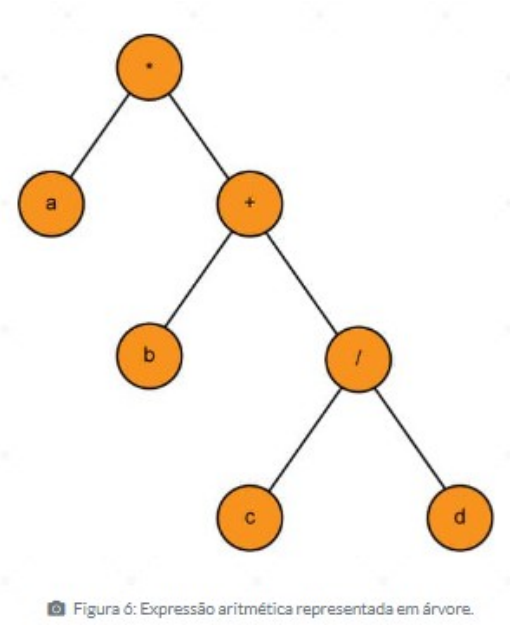


Figura 6: Expressão aritmética representada em árvore.

Relações Hierárquicas

1. **Filhos e Pais**:
   - Nós das subárvores \(T(k) \) de um nó \(k \) são chamados de **filhos** de \(k \).
   - O nó \(k \) é o **pai** de seus filhos.

2. **Irmãos**:
   - Nós que compartilham o mesmo pai são **irmãos** .

3. **Tios e Sobrinhos**:
   - Se \(w \) é filho de \(n_1 \), então \(n_2, n_3, ... \), que são irmãos de \(n_1 \), são **tios** de \(w \).
   - \(w \), por sua vez, é **sobrinho** deles.

4. **Avós e Netos**:
   - Se \(w \) é filho de um nó \(n_1 \), o pai de \(n_1 \) é o **avô** de \(w \).
   - \(w \) é, portanto, o **neto** do nó \(k \).

5. **Descendentes e Ancestrais**:
   - Um nó \(p \) pertencente à subárvore \(T(k) \) é um **descendente** de \(k \).
   - O nó \(k \) é o **ancestral** de \(p \).
   - Se \(p \neq k \), \(p \) é um **descendente próprio** de \(k \), e \(k \) é um **ancestral próprio** de \(p \).
   - Pela definição, um nó é **ancestral** e **descendente** de si mesmo.

Graus, Caminhos e Níveis em Árvores

Graus de Nós e Árvores

1. **Grau de saída de um nó**: Número de filhos do nó.
2. **Classificação dos nós**:
   - **Folha**: Nó sem descendentes próprios.
   - **Interior**: Nó que não é folha.

- **Raiz**: Nó interior sem ancestrais próprios.

3. **Grau de uma árvore**: Maior grau entre os nós.
    - Exemplos:
        - Grau 2: **Árvore binária**.
        - Grau 3: **Árvore ternária**.
        - Grau mmm: **Árvore mmm-ária**.

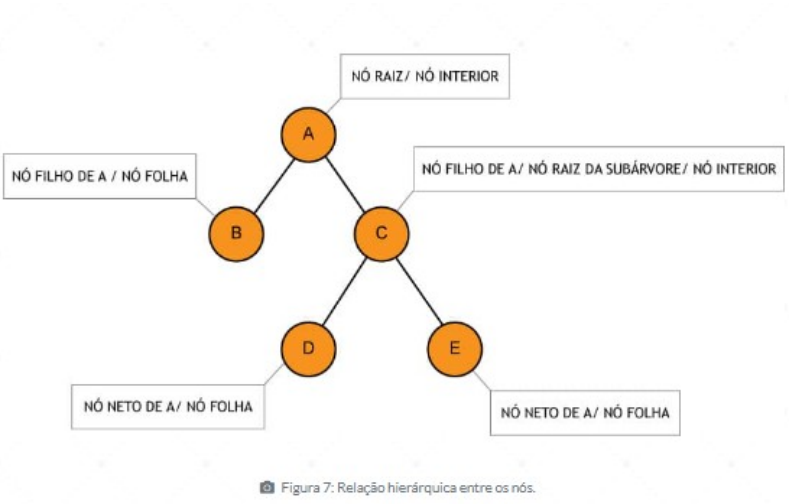


Figura 7: Relação hierárquica entre os nós.

No exemplo da Figura 1:

- **A**: Raiz da árvore.
- **C**: Raiz da subárvore direita de **A**, filho de **A**, irmão de **B**.
- **D**: Neto de **A**.
- **Folhas**: **B, D, E** (grau 0).
- **Interiores**: **A, C** (grau 2).

Caminho ou Percurso

1. **Definição**: Sequência de nós n1,n2,…,nin_1, n_2, \dots, n_in1,n2,…,ni, onde entre nós consecutivos existe relação de **pai** ou **filho**.
    - Exemplo válido: **ACE**.
    - Exemplo inválido: **ABC** (sem relação de ancestralidade entre **B** e **C**).
2. **Comprimento do caminho**: Número de pares consecutivos no percurso: q−1q - 1q−1, onde qqq é o número de nós.

Nível de um Nó

1. **Definição**: Número de nós no caminho entre a raiz e o nó.
    - **Raiz**: Sempre nível 1.
2. **Propriedade**: Cada nó possui **um único caminho** até a raiz devido à disjunção das subárvores.

Essas definições fundamentam a hierarquia e relações em estruturas de árvores.

Altura de Nós, Árvores e Árvores Ordenadas

Altura de um Nó

- **Definição**: Número de nós no maior caminho do nó ttt até um de seus descendentes.
    - **Folhas**: Altura 1.
    - **Exemplo**: Na Figura 1, o nó **A** tem altura 3 (maior caminho até uma folha).
- **Propriedades**:

- Podem existir múltiplos caminhos do nó até folhas, mas cada folha possui um único caminho.
- A altura considera o caminho mais longo.

Altura de uma Árvore

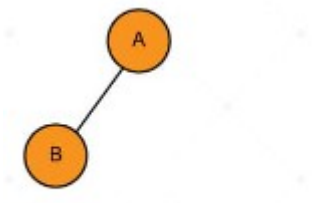
- **Definição**: Maior valor entre os níveis dos nós.
- **Subárvores**: A altura de uma subárvore com raiz vvv, representada por h(v)h(v)h(v), considera apenas os nós pertencentes à subárvore U(v)U(v)U(v).
- **Exemplo (Figura 1)**:
  - Altura da árvore: 3.
  - Grau da árvore: 2 (árvore binária).

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

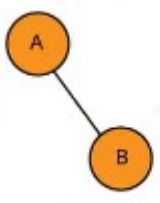
Árvores Ordenadas

- **Definição**: Uma árvore é ordenada quando os filhos de cada nó seguem uma ordem, convencionalmente da esquerda para a direita.
- **Importância**: A posição relativa das subárvores diferencia árvores ordenadas.
- **Exemplo**: Na Figura 1, os nós estão alfabeticamente ordenados. Já as árvores das Figuras 8 e 9 são diferentes devido à posição relativa das subárvores.

Este resumo apresenta os conceitos de altura de nós e árvores, além de árvores ordenadas, fundamentais para entender propriedades e classificações dessas estruturas.



Árvore com a subárvore direita vazia.



9: Árvore com a subárvore esquerda vazia.

Propriedades Básicas de Árvores e Isomorfismo

Número de Folhas em uma Árvore

- Para uma árvore com n>1n > 1n>1 nós:
  - **Mínimo de folhas**: 1.

        - Ocorre quando todos os nós estão em uma mesma linha vertical (relação de ancestralidade), formando uma estrutura linear.
        - Altura da árvore: nnn.
    - **Máximo de folhas**: n−1n - 1n−1.
        - Ocorre quando o nó raiz possui todos os outros n−1n - 1n−1 nós como filhos diretos.
        - Altura da árvore: 2.
- **Exemplo**:
  Para n=5n = 5n=5, uma árvore pode ter:
    - 1 folha (estrutura linear).
    - 4 folhas (estrutura de altura mínima).

Isomorfismo de Árvores

- **Definição**: Duas árvores são **isomorfas** quando podem ser transformadas em estruturas idênticas por meio da permutação na ordem de suas subárvores.
    - Rearranjos acontecem apenas entre **nós irmãos** (mesmo nível).
- **Exemplo**:
    - Figura 1 e Figura 10 são isomorfas, pois a segunda é uma imagem espelhada da primeira.
    - Figura 11 não é isomorfa às Figuras 1 e 10, pois requer alterações nos níveis dos nós (não permitido).
- **Características**:
    - Nem todas as subárvores precisam ser reordenadas para que o isomorfismo ocorra.
    - Em árvores ordenadas, o isomorfismo deve respeitar a sequência definida pela ordenação (ex.: da esquerda para a direita).

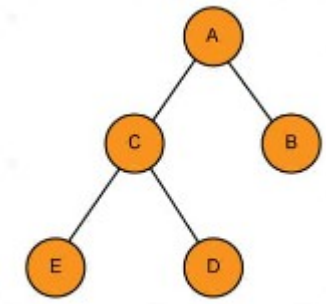
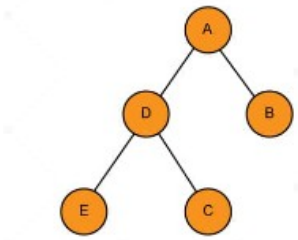


Figura 10: Árvore isomorfa.



ıra 11: Árvore não isomorfa da árvore da Figura 1 ou da Figura

Propriedades do Isomorfismo

1. Para árvores não ordenadas, o isomorfismo se baseia apenas na estrutura e na relação hierárquica entre nós.
2. O rearranjo deve preservar a estrutura global, sendo suficiente para tornar as árvores coincidentes.

Dicas e Observações

- A prova formal das propriedades discutidas é recomendada para consolidar o aprendizado.
- O conceito de isomorfismo é útil para identificar equivalências estruturais em diferentes representações de árvores e na análise de dados hierárquicos.

Representação Computacional de Árvores

Alocação Sequencial (Array)

- **Possibilidade**: Uma árvore pode ser implementada com um vetor, mas essa abordagem não é eficiente por vários motivos.
- **Limitações**:
    1. O espaço de memória é fixo, limitado ao valor definido em tempo de programação, o que impede que a árvore cresça dinamicamente.
    2. A unidimensionalidade do vetor torna a manipulação da árvore menos intuitiva, exigindo cálculos matemáticos complexos para operar e estabelecer as relações hierárquicas entre os nós.
- **Exemplo**: A fórmula usada para determinar a posição do filho direito de um nó em um vetor é dada por pai×2+2\text{pai} \times 2 + 2pai×2+2, onde "pai" é o índice do nó pai no vetor. Isso mostra como a implementação de árvores com vetores pode ser trabalhosa.

Código para inserção de nó filho direito:

```
c
Copiar código
int insere_filho_direita (char chave, int pai) {
    if (arvore[pai] == NULL) {
        printf("Erro: no pai inexistente!");
        return 0; // falha
    } else {
        arvore[(pai * 2) + 2] = chave;
        return 1; // sucesso
    }
}
```

Alocação Encadeada

- **Vantagens**:
    1. A árvore cresce conforme necessário em tempo de execução, e a memória não é desperdiçada quando nós são removidos, pois são desalocados.
    2. A implementação é muito mais simples e direta.
- **Desvantagens**:
    1. Ao usar alocação encadeada, cada nó precisa de um número de ponteiros correspondente ao grau da árvore, mais um campo para armazenar a chave do nó.
    2. O espaço ocupado por cada nó em uma árvore m-ária, que tem uma chave inteira, é dado pela fórmula:
    m×tamanho do ponteiro+tamanho da chave do no´m \times \text{tamanho do ponteiro} + \text{tamanho da chave do nó}m×tamanho do ponteiro+tamanho da chave do no´.

Em resumo, a alocação encadeada é a maneira mais comum e eficaz de implementar árvores, pois oferece maior flexibilidade e simplicidade em comparação com a alocação sequencial, que é mais rígida e difícil de manipular devido à sua dependência de cálculos complexos.

Estruturas de Nós para Árvores

- **Árvore Ternária**:

- Cada nó tem 3 filhos: esquerdo, meio e direito.
- Estrutura em C:

```
c
Copiar código
struct No {
    int chave;
    No *filho_esq;
    No *filho_meio;
    No *filho_dir;
};
```

- **Árvore M-ária (Generalizada)**:
  - Cada nó pode ter até *m* filhos, onde *m* é o grau da árvore.
  - Estrutura em C:

```
c
Copiar código
struct No {
    <tipo> chave;
    No *filho_1;
    No *filho_2;
    [...]
    No *filho_m;
};
```

Essas estruturas definem como os nós são organizados nas árvores m-árias e ternárias, permitindo flexibilidade conforme o grau da árvore.

Função de Inserção de Nó na Árvore (Alocação Encadeada)

O **Código 4** apresenta uma função para inserir um nó na subárvore direita de um nó pai, utilizando alocação encadeada:

```
c
Copiar código
int insere_filho_direita (No novo_no, No *pai) {
    if (pai == NULL) {
        printf("Erro: noh pai inexistente!");
        return 0; // falha
    } else
        pai -> filho_dir = novo_no;
    return 1; // sucesso
}
```

- **Comparação com Código 1**:
  - O **Código 4** é mais simples e eficiente, pois não requer cálculos aritméticos. A operação é limitada a um acesso à memória e uma escrita.
  - **Código 1** envolve cálculos para determinar a posição do nó no vetor.
- **Sobrescrita de Nó**:
  - No **Código 1**, sobrescrever um nó é automático pela posição no vetor.
  - No **Código 4**, para sobrescrever, deve-se considerar se o nó tem filhos. Duas abordagens possíveis para essa situação são:
    1. Atualizar apenas a chave do nó, se ele tiver filhos.

2. Transferir os filhos para o novo nó e depois inserir na árvore.

- **Decisão Técnica**: A escolha entre alocação sequencial (vetor) e encadeada depende do problema a ser resolvido, considerando as vantagens e desvantagens de cada abordagem.

O código aborda uma inserção simples, mas destaca a flexibilidade e os ajustes necessários quando ocorre a sobrescrita de nós.

Resumo Detalhado: Árvore Binária

A **árvore binária** é um tipo especial de árvore onde cada nó pode ter no máximo dois filhos. Ela é amplamente utilizada devido às suas propriedades e características.

Definição Formal:
1. Uma **árvore binária**, denotada \(T \), é um conjunto finito de nós com as seguintes características:
  - Se \(T = \emptyset \), a árvore é **vazia**.
  - Caso contrário, existe:
    - Um elemento especial chamado **nó raiz** (\(rT \)).
    - Dois subconjuntos disjuntos chamados **subárvore esquerda** (\(TErT \)) e **subárvore direita** (\(TDrT \)), que também são árvores binárias.

2. **Particularidades**:
  - Um nó pode ter:
    - **0 filhos**: ambas as subárvores vazias.
    - **1 filho**: apenas uma subárvore não vazia.
    - **2 filhos**: ambas as subárvores não vazias.

3. A definição é recursiva: cada subárvore de um nó é também uma árvore binária, podendo ser vazia ou possuir suas próprias subárvores.

Notação:
- Para a **raiz** da árvore, usa-se \(rT \).
- Para a **subárvore esquerda** e **direita** da raiz:
  - Subárvore esquerda: \(TErT \).
  - Subárvore direita: \(TDrT \).
- Para um nó genérico \(v \), usa-se:
  - Subárvore esquerda: \(TEv \).
  - Subárvore direita: \(TDv \).

Propriedades Exclusivas:

- Em árvores binárias, a **posição das subárvores** (esquerda ou direita) é crucial, pois define unicamente sua estrutura.
- Os filhos de um nó são denominados de acordo com sua posição:
  - O nó raiz da **subárvore esquerda** é o **filho esquerdo**.
  - O nó raiz da **subárvore direita** é o **filho direito**.

Resumo Visual:
- Uma árvore binária é composta por:
  - **Raiz** (\(rT \)).
  - **Subárvores** (\(TErT \) e \(TDrT \)), que também seguem a definição de árvore binária.

Essa estrutura recursiva e suas propriedades tornam as árvores binárias fundamentais para diversas aplicações em computação.

Diferença entre Árvores Binárias e Definição de Árvores Estritamente Binárias

Figuras 12(a) e 12(b):
- Ambas são **isomorfas** e **idênticas se analisadas como **árvores ordenadas**, porque o ordenamento entre os nós irmãos é o mesmo.
- São **distintas** quando consideradas **binárias**, pois, em árvores binárias, a **posição** das subárvores (esquerda ou direita) é relevante:
  - Na Figura 12(a): \(TEB \) (subárvore esquerda de \(B \)) é vazia, e \(TDB \) (subárvore direita de \(B \)) contém o nó \(D \).
- Na Figura 12(b): a situação se inverte, com \(D \) na subárvore esquerda.

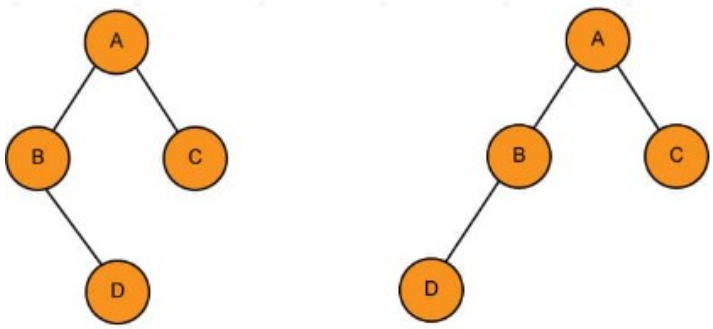


Figura 12: Árvores binárias.

Árvores Binárias vs. Árvores Estritamente Binárias:
- **Árvore binária**: Cada nó pode ter 0, 1 ou 2 filhos.
- **Árvore estritamente binária**:
  - Cada nó possui **exatamente 0 ou 2 filhos**.
  - Nós sem filhos são obrigatoriamente **folhas**.
  - Qualquer árvore binária com pelo menos um nó de apenas um filho **não é estritamente binária**.

Características da Árvore Estritamente Binária:
- Todo nó **interior** (não folha) possui **dois filhos**.
- Estruturalmente, é mais restritiva que uma árvore binária comum.

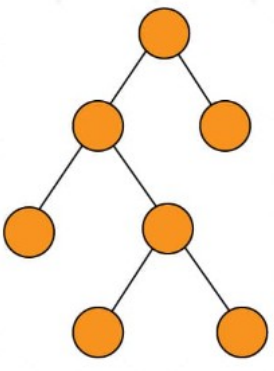


Figura 13: Árvore estritamente binária.

Tipos de Árvores Binárias

- **Árvore Binária Cheia**:
  - Os nós com subárvores vazias estão **somente no último nível**.
  - **Exemplo**: Figura 14.
- **Árvore Binária Completa**:
  - Os nós com subárvores vazias estão no **último ou penúltimo nível**.
  - Toda árvore binária cheia é completa, mas o contrário **não é válido**.
  - **Exemplo**: Figura 15.
- **Árvore Zigue-Zague**:
  - Todos os nós interiores possuem **exatamente uma subárvore vazia**.
  - Não há nós interiores com duas subárvores não vazias.
  - **Exemplo**: Figura 16.

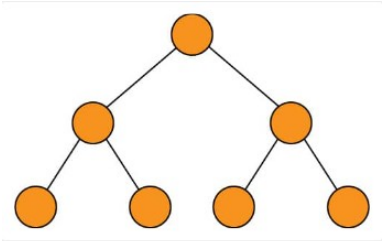


Figura 14: Árvore binária cheia.

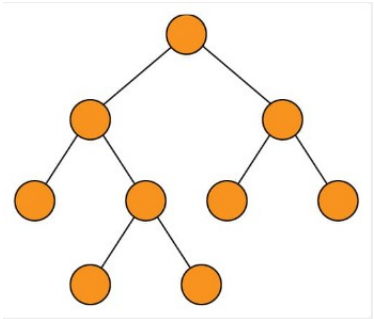


Figura 15: Árvore binária completa.

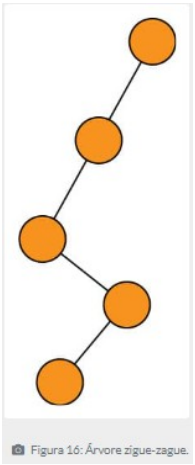


Figura 16: Árvore zigue-zague.

Propriedades de Árvores Binárias

Altura de uma Árvore Binária

- **Altura Máxima**:
  A altura máxima de uma árvore binária ocorre quando cada nó possui exatamente uma subárvore vazia, formando uma árvore zigue-zague (ex.: Figura 16). Nesse caso, todos os **n** nós estão alinhados verticalmente, resultando em altura **h = n**.
- **Altura Mínima**:
  A altura mínima ocorre em uma **árvore binária completa**, onde os níveis são preenchidos antes de iniciar um novo. Nessa estrutura:
  - Somente os nós do penúltimo e último níveis podem ter subárvores vazias (ex.: Figura 15).
  - Nenhum rearranjo de nós pode reduzir a altura sem contrariar a definição de árvore completa.
  - A altura mínima de uma árvore binária completa com **n** nós é: $h=1+\lfloor \log 2n \rfloor$

  Assim, para qualquer árvore binária **T** com **n > 0** nós, sua altura **h** está no intervalo:

  $n \leq h \leq 1+\lfloor \log 2n \rfloor$

Número de Subárvores Vazias

Em árvores binárias, cada nó possui duas posições possíveis para subárvores (esquerda e direita). Assim:

- Para **n = 1** (nó raiz e folha):
  Há **2 subárvores vazias** (esquerda e direita do nó raiz).
- Para **n = 2** (nó raiz com um filho):

- O nó raiz tem uma subárvore vazia e um filho.
- O nó filho, sendo folha, tem **2 subárvores vazias**.
  Total: **3 subárvores vazias**.

Por indução, pode-se provar que, para qualquer **n > 0**, o número de subárvores vazias é:

n+1

Comentários Importantes

- O **grau da árvore** (número máximo de filhos de qualquer nó) é sempre **≤ 2**, mas o grau total depende da disposição dos nós.
- Mesmo árvores zigue-zague (grau máximo 1) são consideradas binárias se respeitarem a definição de no máximo dois filhos por nó.

Essas propriedades ajudam a compreender como a estrutura das árvores binárias influencia diretamente seu comportamento, uso e eficiência em algoritmos e dados.

Percurso em Árvores Binárias: Resumo

Definição e Importância

- **Percorrer uma árvore binária** significa visitar sistematicamente todos os seus nós.
- Árvores binárias são amplamente utilizadas, como em indexadores, tornando eficiente o percurso essencial para aplicações práticas.

Conceito de Visitar um Nó

- **Visitar um nó** envolve realizar uma operação sobre a informação armazenada nele, como imprimir ou atualizar seu conteúdo.
- **Não é visitação** acessar campos do tipo ponteiro de um nó (ex.: buscar o endereço de um nó filho).

Estrutura do Percurso

- Uma árvore binária genérica (Figura 17) possui:
  - Um nó raiz (**r**).
  - Duas subárvores: esquerda (**TEr**) e direita (**TDr**).
- Para percorrer a árvore:
  - É necessário visitar todos os nós, incluindo os das subárvores esquerda e direita do nó raiz.

Uso da Recursividade

- A recursão surge naturalmente no percurso de árvores, pois:
  - Para visitar uma árvore, é necessário percorrer recursivamente suas subárvores.
  - O processo continua até encontrar uma subárvore vazia, momento em que a recursão é interrompida.

Conclusão

Compreender o percurso e a recursividade é fundamental para implementar algoritmos eficazes em árvores binárias.

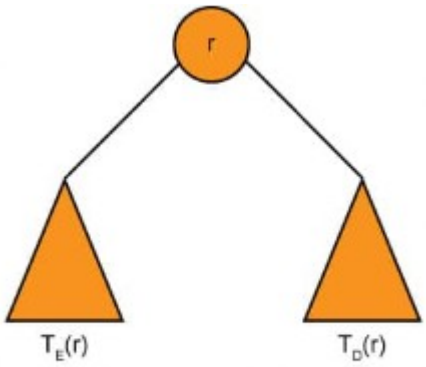


Figura 17: Árvore binária genérica com raiz r.

Percursos em Árvores Binárias: Resumo Detalhado

Impacto da Ordem no Percurso

- A ordem das operações de percorrer a subárvore esquerda, percorrer a subárvore direita e visitar o nó define o tipo de percurso realizado.
- Cada tipo de percurso é adequado para resolver problemas específicos. Há três combinações possíveis: **pré-ordem**, **ordem simétrica** e **pós-ordem**.

Tipos de Percursos

1. Percurso em Pré-Ordem

- **Ordem das operações:**
    1. Visitar o nó.
    2. Percorrer recursivamente a subárvore esquerda.
    3. Percorrer recursivamente a subárvore direita.
- **Exemplo:** Aplicando o algoritmo à árvore da **Figura 1**, a sequência de saída será: **A B C D E**.
- **Código em C:**

```
c
Copiar código
void pre_ordem(No *ptr) {
    printf(ptr->chave); // visita o nó
    if (ptr->filho_esquerda != NULL)
        pre_ordem(ptr->filho_esquerda); // percorre subárvore esquerda
    if (ptr->filho_direita != NULL)
        pre_ordem(ptr->filho_direita); // percorre subárvore direita
}
```

---

2. Percurso em Ordem Simétrica

- **Ordem das operações:**
    1. Percorrer recursivamente a subárvore esquerda.
    2. Visitar o nó.
    3. Percorrer recursivamente a subárvore direita.
- **Exemplo:** Aplicando o algoritmo à árvore da **Figura 1**, a sequência de saída será: **B A D C E**.
- **Código em C:**

```
c
Copiar código
void ordem_simetrica(No *ptr) {
    if (ptr->filho_esquerda != NULL)
        ordem_simetrica(ptr->filho_esquerda); // percorre subárvore
esquerda
    printf(ptr->chave); // visita o nó
    if (ptr->filho_direita != NULL)
        ordem_simetrica(ptr->filho_direita); // percorre subárvore direita
}
```

---

3. Percurso em Pós-Ordem

- **Ordem das operações:**
  1. Percorrer recursivamente a subárvore esquerda.
  2. Percorrer recursivamente a subárvore direita.
  3. Visitar o nó.
- **Exemplo:** Aplicando o algoritmo à árvore da **Figura 1**, a sequência de saída será: **B D E C A**.
- **Código em C:**

```
c
Copiar código
void pos_ordem(No *ptr) {
    if (ptr->filho_esquerda != NULL)
        pos_ordem(ptr->filho_esquerda); // percorre subárvore esquerda
    if (ptr->filho_direita != NULL)
        pos_ordem(ptr->filho_direita); // percorre subárvore direita
    printf(ptr->chave); // visita o nó
}
```

Análise dos Algoritmos

- **Semelhanças:** Os três algoritmos têm estruturas semelhantes e são baseados em recursividade.
- **Diferenciação:** A diferença principal está na ordem em que as operações (visitar e percorrer subárvores) são realizadas.
- **Extrapolação:** Compreender um dos códigos (como o de **ordem simétrica**) facilita a adaptação para os outros métodos.

Código de Ordem Simétrica (detalhado):

```
c
Copiar código
void ordem_simetrica(No *ptr) {
    if (ptr->filho_esquerda != NULL)
        ordem_simetrica(ptr->filho_esquerda); // percorre subárvore esquerda
    printf(ptr->chave); // visita o nó
    if (ptr->filho_direita != NULL)
        ordem_simetrica(ptr->filho_direita); // percorre subárvore direita
}
```

Os três percursos são fundamentais em algoritmos que manipulam árvores binárias, sendo amplamente usados em diversas aplicações, como indexação de dados e análise de estruturas.

Funcionamento dos Algoritmos de Percurso em Árvores Binárias

1. **Recebimento de Parâmetros:**
    - Os algoritmos de percurso recebem um ponteiro para o nó raiz da árvore binária na primeira chamada.
    - Não é testado se o nó raiz é nulo, pois percorrer uma árvore vazia não faz sentido. Esse teste deve ser feito pelo código chamador.
2. **Funcionamento Geral:**
    - Cada função verifica se a subárvore esquerda é vazia:
        - Se **não for vazia**, chama-se recursivamente o algoritmo com o ponteiro para a raiz da subárvore esquerda.
    - Após tratar a subárvore esquerda, realiza-se a **visitação do nó** (exemplo: imprimir a informação armazenada no nó).
    - Verifica-se, então, a existência da subárvore direita:
        - Se **não for vazia**, chama-se recursivamente o algoritmo com o ponteiro para a raiz da subárvore direita.
    - Este processo é idêntico nos três tipos de percursos (pré-ordem, ordem simétrica e pós-ordem).
3. **Recursão e Subárvores Vazias:**
    - As chamadas recursivas só são realizadas se as subárvores não forem vazias. Uma subárvore vazia impede a recursão.
4. **Flexibilidade dos Algoritmos:**
    - Nos exemplos apresentados, a visitação corresponde à impressão da informação no nó.
    - Os algoritmos podem ser adaptados para outras finalidades, como retornar endereços de nós ou realizar atualizações nos dados.

Esses princípios garantem simplicidade e generalidade na aplicação dos algoritmos de percurso em diferentes contextos.

Observações e Generalizações sobre Algoritmos de Percurso em Árvores

1. **Chamadas Recursivas:**
    - Em uma árvore com **n nós**, os algoritmos de percurso fazem **n chamadas**, uma por nó.
    - Um nó pode ser acessado mais de uma vez para visitar sua subárvore direita, mas é **visitado apenas uma vez**.
2. **Implementações Não Recursivas:**
    - Embora possíveis, são mais complexas porque exigem controle explícito da direção (esquerda ou direita) e dos ramos já percorridos.
    - A recursividade simplifica a implementação, tornando-a mais intuitiva.

3. **Generalização para Árvores m-árias:**
    - Os percursos em árvores binárias podem ser facilmente adaptados para árvores m-árias:
        - **Pré-ordem:** Visita-se o nó antes de percorrer todas as subárvores recursivamente.
        - **Pós-ordem:** Percorrem-se todas as subárvores recursivamente antes de visitar o nó.
        - **Ordem Simétrica:** Desafio adicional é posicionar a visita do nó entre os percursos das subárvores.
4. **Outros Algoritmos:**
    - Embora os percursos em pré-ordem, pós-ordem e ordem simétrica sejam os mais fundamentais, outras variações podem ser criadas.
5. **Exemplo: Cálculo da Altura da Árvore (Código 8):**
    - A altura da árvore pode ser calculada substituindo a operação de visita do nó por um cálculo do nível:
        - **Altura de subárvores:** Determinada para os filhos esquerdo e direito (ou zero, se vazios).
        - **Altura do nó:** Maior altura entre as subárvores somada a 1.
    - Requer um campo adicional nos nós chamado "altura" para armazenar o resultado.
    - **Variáveis estáticas** são usadas para armazenar temporariamente as alturas das subárvores.

Código 8: Implementação para Cálculo da Altura

```
c
Copiar código
static void altura ( No *ptr ) {
    if (ptr -> filho_esquerda != NULL)
        altura_subarvore_esquerda = ptr -> filho_esquerda -> altura;
    else
        altura_subarvore_esquerda = 0;

    if (ptr -> filho_direita != NULL)
        altura_subarvore_direita = ptr -> filho_direita -> altura;
    else
        altura_subarvore_direita = 0;

    if (altura_subarvore_esquerda > altura_subarvore_direita)
        ptr -> altura = altura_subarvore_esquerda + 1;
    else
        ptr -> altura = altura_subarvore_direita + 2;
}
```

6. **Aplicação Futura:**
    - Na próxima seção, serão introduzidos tipos específicos de árvores usadas para pesquisa, aplicadas a problemas fundamentais de computação.

Árvores de Pesquisa

1. **Objetivo e Importância:**

- Árvores são aplicadas para **resolver problemas de busca** em conjuntos de elementos identificados por **chaves únicas**.
- A busca consiste em localizar um elemento no conjunto através de sua chave.
- Ao usar árvores, elementos e suas chaves são organizados nos nós, permitindo acesso eficiente.

2. **Conceito de Chave:**
    - A chave é usada como **indexador** do conjunto e está associada a um elemento.
    - Recuperar a chave implica recuperar o elemento correspondente.
3. **Aplicações Práticas:**
    - Árvores são empregadas em:
        - **Bancos de dados** para indexação eficiente.
        - **Sistemas operacionais**, como na gestão de arquivos e alocação de espaço em disco.
        - Diversas outras situações que requerem organização e acesso rápido.
4. **Relevância do Balanceamento:**
    - A eficiência das operações em árvores depende de sua **estrutura balanceada**.
    - O estudo de técnicas de balanceamento é essencial para otimizar o desempenho das buscas e outras operações.

Esta seção foca no uso prático de árvores, destacando sua aplicação em problemas de busca e a importância do balanceamento para operações eficientes.

Árvore Binária de Busca (ABB)

1. **Definição de ABB:**
    - Uma ABB é uma **árvore binária rotulada** com regras específicas para organização de chaves.
    - Cada nó corresponde a uma única chave do conjunto $C=\{c1,c2,...,cn\}$, onde $c1<c2<...<cn$.
    - As regras de rotulagem:
        - **Subárvore esquerda**: Todos os nós têm rótulos menores que o nó pai.
        - **Subárvore direita**: Todos os nós têm rótulos maiores que o nó pai.
2. **Problema de Busca Formalizado:**
    - Dado um conjunto C, o objetivo é:
        - Identificar o índice nnn correspondente à chave procurada, ou
        - Determinar que a chave não pertence ao conjunto.
3. **Características Importantes:**
    - A ABB organiza as chaves de forma ordenada, facilitando a busca.
    - Para um conjunto C, podem existir **múltiplas ABBs válidas**, desde que as regras sejam respeitadas.
    - Chaves numéricas são comuns, mas chaves alfabéticas também podem ser usadas se houver relação de ordem (ex.: $A<B<C$).
4. **Impacto da Estrutura:**

- A ordenação dos nós da ABB segue a regra:
    - **Subárvore esquerda**: Contém valores menores ou é inexistente.
    - **Subárvore direita**: Contém valores maiores ou é inexistente.
- Essa estrutura permite que a ABB resolva eficientemente problemas de busca.

5. **Exemplo:**

    - Para o conjunto C={1,2,3,4,5}, é possível construir mais de uma ABB válida, como ilustrado na Figura 18.

As ABBs são essenciais para resolver problemas de busca devido à sua organização eficiente das chaves.

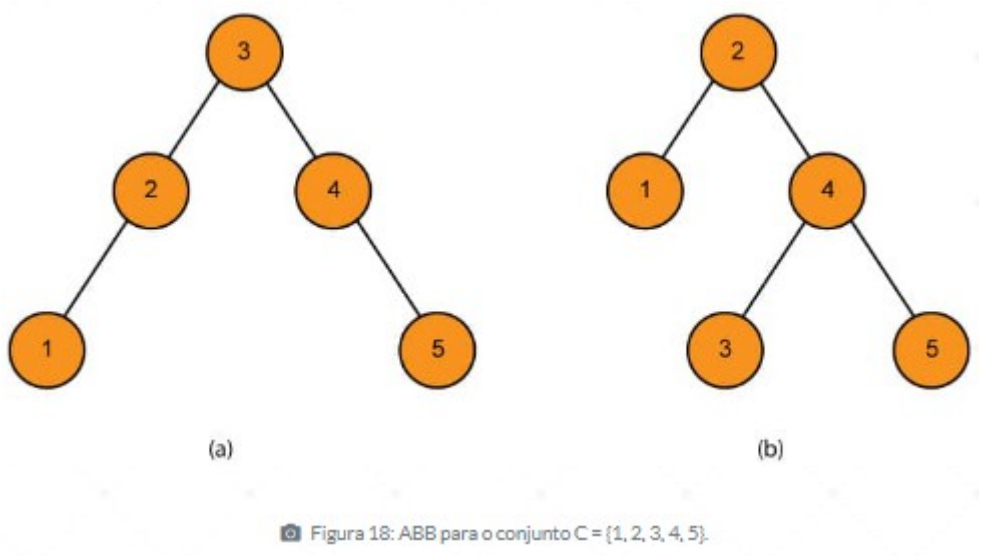


Figura 18: ABB para o conjunto C = {1, 2, 3, 4, 5}.

Busca em Árvores Binárias de Busca (ABB)

Funcionamento da Busca em ABB

1. **Exemplo de Busca**:

    - Procurando a chave **3** na árvore "b" da Figura 18:
        - **Comparações realizadas**:
            - Compara com a raiz (**2**) → vai para a subárvore direita.
            - Compara com o nó (**4**) → vai para a subárvore esquerda.
            - Encontra a chave no nó (**3**).
        - Foram feitas 3 comparações, enquanto na árvore "a" da mesma figura, apenas 1 seria necessária.

2. **Considerações Importantes**:

    - A **estrutura da ABB** influencia o número de comparações.
    - No **pior caso**, uma árvore degenerada (zigue-zague) exige nnn comparações (igual ao número de nós).
    - O **balanceamento** da ABB é crucial para eficiência, pois árvores de altura mínima maximizam o desempenho.

3. **Busca de Chaves Inexistentes**:

    - Ao buscar **6** na árvore "b":
        - Segue até o nó **5**, cuja subárvore direita é vazia.
        - Conclusão: chave não pertence ao conjunto.

Código de Busca em ABB

```
c
Copiar código
int busca_arvore (int chave, No *ptr) {
    if (ptr->chave == chave)
        return 1; // chave encontrada
    else if (chave < ptr->chave) {
        if (ptr->filho_esquerdo == NULL)
            return 0; // chave não encontrada
        else
            return busca_arvore(chave, ptr->filho_esquerdo);
    } else {
        if (ptr->filho_direito == NULL)
            return 0; // chave não encontrada
        else
            return busca_arvore(chave, ptr->filho_direito);
    }
}
```

Altura de ABB e Impacto na Busca

1. **Relação entre Nós (n) e Altura (h)**:
    - Para uma árvore binária **cheia**:
        - $n=2^h-1$ (número de nós dobra a cada nível).
    - Para uma árvore binária **completa**:
        - $2^{h-1} \leq n \leq 2^h-1$.
    - ABBs de **altura mínima** tornam as buscas mais eficientes.
2. **Construção da ABB**:
    - Inserir chaves na ordem $C=\{1,2,3,4,5,6,7\}$:
        - Gera uma **árvore zigue-zague** com altura máxima.
    - Usar a **chave média como raiz** (como na árvore “a” da Figura 18):
        - Distribui as chaves de forma equilibrada, produzindo uma árvore binária completa.

Objetivo

Construir ABBs de forma que sejam árvores binárias completas, maximizando eficiência na busca.

Árvores Balanceadas

Motivação e Conceito

- **Problema**: Inserções e remoções em árvores binárias de busca (ABB) podem degenerá-las em estruturas ineficientes, como uma árvore zigue-zague. Isso aumenta o custo de busca.
- **Objetivo**: Garantir que a árvore mantenha desempenho eficiente mesmo após alterações, minimizando o número de comparações necessárias.
- **Árvores Balanceadas**:
    - Mantêm o custo de acesso às chaves próximo ao ótimo, ou seja, proporcional à altura mínima da árvore.
    - Não precisam ser completas, mas devem evitar degeneração.

Árvore Completa e Eficiência

- Árvores completas minimizam o número de comparações ($1+\lfloor \log n \rfloor$) e são um modelo ideal para eficiência de busca.
- Porém, manter a árvore sempre completa pode ser inviável devido ao custo elevado de balanceamento após inserções ou remoções.

Relaxamento do Balanceamento

- Em vez de exigir eficiência máxima ($1+\lfloor \log n \rfloor$), permite-se um desempenho levemente pior, mas ainda dentro da mesma ordem de grandeza ($O(\log n)$).
- **Requisitos de Balanceamento**:
    - Altura da árvore deve ser proporcional à altura de uma árvore completa com o mesmo número de nós.
    - A limitação se aplica recursivamente às subárvores, garantindo estrutura equilibrada em toda a árvore.

Vantagens do Relaxamento

- Reduz o custo do balanceamento sem comprometer significativamente a eficiência da busca.
- Permite manter a árvore viável para problemas práticos, mesmo com operações frequentes de inserção e remoção.

Próximo Passo

- Estudo de árvores AVL, um exemplo de árvore balanceada que implementa essas ideias.

Árvore AVL

Definição

- **Árvore AVL**: Tipo de árvore binária de busca balanceada onde:
    1. As alturas das subárvores esquerda e direita de cada nó diferem por no máximo 1 unidade.
    2. As subárvores esquerda e direita também são árvores AVL.

Conceitos Chave

- **Nó Regulado**: Quando a diferença de altura entre as subárvores de um nó é ≤ 1.
- **Nó Desregulado**: Quando essa diferença excede 1.
- A definição é recursiva: cada subárvore deve seguir as mesmas regras, garantindo que o balanceamento se propague por toda a árvore.

Propriedades

- Toda **árvore completa** é uma árvore AVL, pois as diferenças de altura em qualquer nó são ≤ 1.
- Nem toda árvore AVL é completa, pois podem existir subárvores vazias em níveis mais altos, como no antepenúltimo nível.

Exemplo

- A Figura 19 ilustra uma árvore AVL que não é completa, destacando a flexibilidade dessa estrutura em relação ao balanceamento.

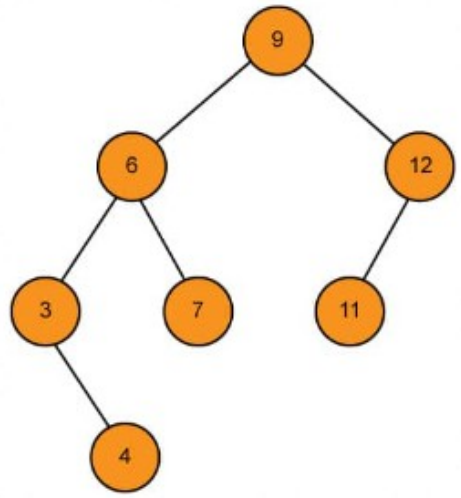


Figura 19: Árvore AVL.

Importância

Árvores AVL são projetadas para manter eficiência em buscas, mesmo após inserções e remoções, garantindo que a altura permaneça aproximadamente O(logn).

Balanceamento em Árvores AVL

Altura e Número Mínimo de Nós

- Em uma **árvore AVL**, a diferença máxima de altura entre as subárvores de qualquer nó é 1.
- Para uma árvore AVL de altura hhh, a maior subárvore da raiz terá altura h−1h-1h−1, enquanto a menor poderá ter h−2h-2h−2 (no caso de configuração com o menor número de nós).
- Aplicando esse raciocínio recursivamente, é possível construir uma árvore AVL de altura hhh com o menor número de nós possível.
- A relação entre a altura hhh e o número de nós nnn em uma árvore AVL garante que hhh seja da ordem de logn\log nlogn, o que mantém a eficiência nas buscas.

Processo de Balanceamento

- Durante a **inserção de um nó**, a estrutura da árvore pode ser alterada, potencialmente desregulando algum nó.
- Após cada inserção, verifica-se toda a árvore para identificar nós desregulados.
    - **Se não houver desregulação**: nenhuma ação é necessária.
    - **Se houver desregulação**: é preciso realizar ajustes para restaurar o balanceamento.

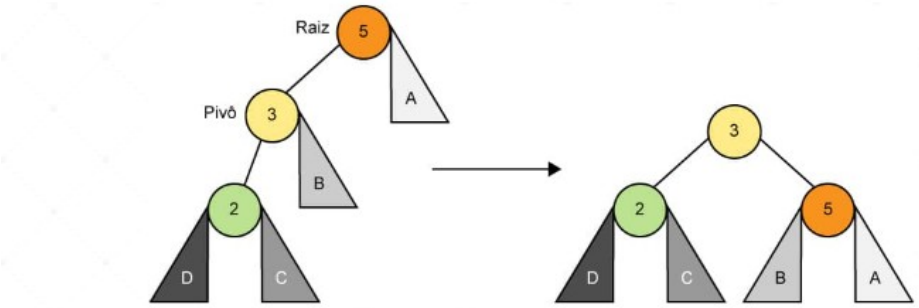
Técnicas de Balanceamento

Para corrigir a desregulação em uma árvore AVL, são aplicadas **quatro tipos de rotações**:

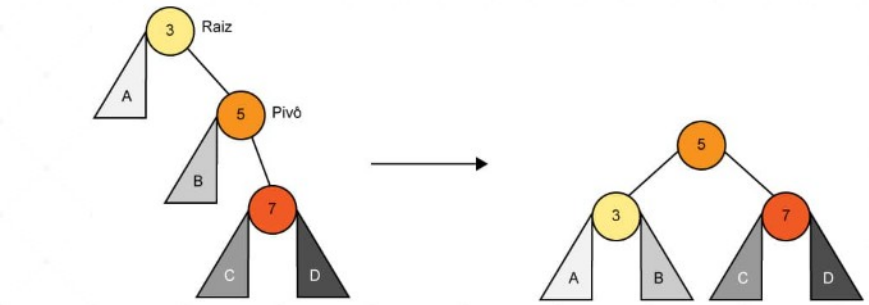
1. **Rotação simples à direita**: Corrige desbalanceamento causado por inserção na subárvore esquerda do filho esquerdo do nó desregulado.
2. **Rotação simples à esquerda**: Corrige desbalanceamento causado por inserção na subárvore direita do filho direito do nó desregulado.
3. **Rotação dupla à direita**: Corrige desbalanceamento causado por inserção na subárvore direita do filho esquerdo do nó desregulado.
4. **Rotação dupla à esquerda**: Corrige desbalanceamento causado por inserção na subárvore esquerda do filho direito do nó desregulado.

Essas rotações reorganizam os nós da árvore, mantendo a propriedade AVL e garantindo que a eficiência da busca não seja comprometida.

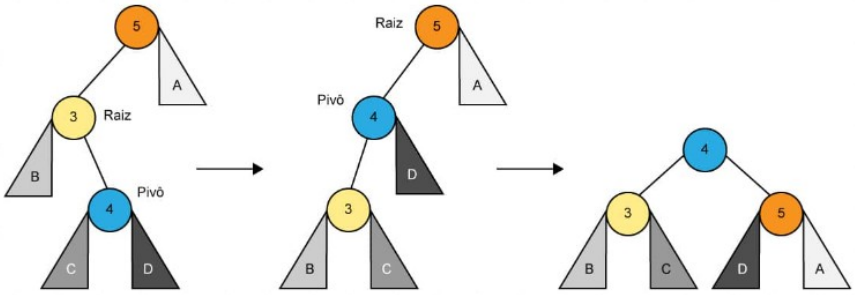
Rotação simples à direita.



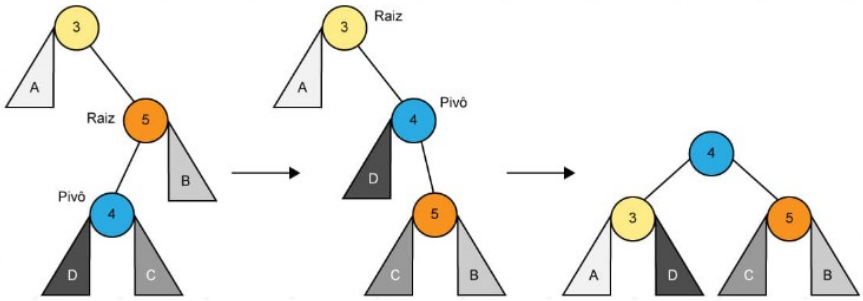
Rotação simples à esquerda.



Rotação dupla à direita.



Rotação dupla à esquerda.



Balanceamento de Árvores AVL após Inserção

- **Inserção e Desbalanceamento**: Após a inserção de um nó q em uma árvore AVL T, a árvore pode se desbalancear. Se a inserção não causar desbalanceamento, a árvore permanece AVL. Caso contrário, é necessário balancear os nós desregulados.
- **Nó Desregulado mais Próximo**: O primeiro nó p desregulado após a inserção de q será o mais próximo das folhas e estará no caminho entre q e a raiz.
- **Análise das Alturas**: Definem-se hEp e hDp como as alturas das subárvores esquerda e direita de p, respectivamente. A diferença de altura entre as subárvores de p será $|hEp - hDp| = 2$, indicando o desbalanceamento.

- **Transformações para Balanceamento**:
    1. **Rotação à direita**: Se a inserção ocorrer na subárvore esquerda do filho esquerdo de ppp (com hEp>hDp e hEu>hDu).
    2. **Rotação dupla à direita**: Se a inserção ocorrer na subárvore direita do filho esquerdo de ppp (com hEp>hDp e hEu<hDu).
    3. **Rotação à esquerda**: Se a inserção ocorrer na subárvore direita do filho direito de ppp (com hEp<hDph e hEz<hDz).
    4. **Rotação dupla à esquerda**: Se a inserção ocorrer na subárvore esquerda do filho direito de ppp (com hEp<hDp e hEz>hDz).
- **Importância das Árvores AVL**: Embora as árvores AVL sejam um exemplo importante de árvores balanceadas, existem outras estruturas que seguem o mesmo conceito. Entender o funcionamento das árvores AVL é essencial para o uso eficiente de árvores binárias de busca.

Um **vetor** é uma estrutura de dados ou conceito matemático que pode ser entendido de diferentes formas dependendo do contexto:

Em Programação e Computação

Um vetor é uma coleção ordenada de elementos do mesmo tipo, armazenados de forma contígua na memória. Ele é usado para representar um conjunto de dados relacionados e permite o acesso rápido aos seus elementos pelo índice.

- **Características:**
    1. **Elementos homogêneos**: Todos os elementos têm o mesmo tipo.
    2. **Acesso por índice**: Cada posição do vetor é identificada por um número inteiro chamado índice, começando normalmente de 0.
    3. **Tamanho fixo**: Geralmente, o tamanho do vetor é definido no momento de sua criação (em algumas linguagens, isso pode variar).
- **Exemplo em Python:**

```
python

vetor = [1, 2, 3, 4, 5]  # Um vetor de inteiros
print(vetor[2])  # Acessa o terceiro elemento (índice 2): saída será 3
```

No código **`printf("Conteudo da variavel v_num: %d\n", v_num);`**, o **`%d`** é um **especificador de formato** que indica ao `printf` que o valor a ser exibido corresponde a um número inteiro decimal.

Explicação detalhada:

1. **`printf`**: É uma função usada em linguagens como C para exibir texto e valores na saída padrão (geralmente o console).
2. **`%d`**:

- É o especificador de formato usado para exibir valores do tipo **inteiro decimal** (`int`).
- Ele instrui o `printf` a procurar, na lista de argumentos após a string formatada, um valor inteiro correspondente a ser exibido no local onde o **`%d`** aparece.

3. **`v_num`**:
   - É a variável cujo valor será inserido no lugar do **`%d`**.
   - Para que funcione corretamente, o tipo de `v_num` deve ser **int** (ou algo compatível, como `short` ou `long`, dependendo do compilador e da plataforma).
4. **`\n`**:
   - Representa uma nova linha, fazendo com que o cursor de saída seja movido para a linha seguinte após a exibição.

Funcionamento completo do exemplo:

Se `v_num` tiver o valor **42**, o código:

```c
int v_num = 42;
printf("Conteudo da variavel v_num: %d\n", v_num);
```

Produzirá a seguinte saída no console:

```yaml
Conteudo da variavel v_num: 42
```

Resumo de outros especificadores relacionados ao `%d`:

Especificador	Significado	Tipo esperado
`%d`	Inteiro decimal com sinal	`int`
`%u`	Inteiro decimal sem sinal	`unsigned int`
`%x`	Inteiro em hexadecimal (minúsculas)	`unsigned int`
`%X`	Inteiro em hexadecimal (maiúsculas)	`unsigned int`
`%o`	Inteiro em octal	`unsigned int`

Nota:

O uso correto do especificador de formato é essencial. Se o tipo da variável não for compatível com o especificador usado (por exemplo, um `float` no lugar de um `int` com `%d`), o comportamento pode ser imprevisível.

Matriz é uma maneira de representar dados dividindo-os em linhas e colunas. $A = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

Vou enviar o conteúdo de uma página de uma apostila de estudo. Preciso de um resumo claro e direto que mantenha as informações principais e mais relevantes. Ao resumir, siga estas instruções: Extraia as ideias centrais e informações essenciais. Mantenha o foco nos conceitos, definições, e fatos importantes que são necessários para uma revisão eficaz. Eu não quero que você simplifique as coisas ao ponto de deixar o mínimo de informações possível, não é esse o objetivo aqui, o objetivo é pegar o texto original, e simplificá-lo, tirando quaisquer redundâncias, repetições, ou palavras desnecessárias. Se algo pode ser dito eficientemente com 5 palavras, não há necessidade de usar 10. Texto a ser resumido: