



ESTRUTURA DE DADOS

PIETRO MARTINS DE OLIVEIRA

ROGÉRIO DE LEON PEREIRA

ORGANIZADORA

TATIANA ALENCAR

ACESSE AQUI ESTE
MATERIAL DIGITAL!

EXPEDIENTE

Coordenador(a) de Conteúdo

Nader Ghoddosi

Projeto Gráfico e Capa

Arthur Cantareli Silva

Editoração

Adrian Marcareli dos Santos, Lucas
Pinna Silveira Lima, Lavignia da Silva
Santos, Caroline Casarotto Andujar e
Camila Luiza Nardelli

Design Educacional

Amanda Peçanha

Revisão Textual

Erica F. Ortega

Ilustração

Eduardo Aparecido Alves, Bruno Cesar
Pardinho Figueiredo e Andre Luis
Azevedo da Silva.

Fotos

Shutterstock

FICHA CATALOGRÁFICA

C397 Centro Universitário Leonardo da Vinci.

Núcleo de Educação a Distância. **OLIVEIRA**, Pietro Martins de;
PEREIRA, Rogério de Leon.

Estrutura de Dados / Pietro Martins de Oliveira, Rogério de Leon
Pereira; organizador: Tatiana Alencar. - Indaial, SC: Arquê, 2023.

284 p.

ISBN papel 978-65-6083-448-4

ISBN digital 978-65-6083-449-1

"Graduação - EaD".

1. Programação 2. Dados 3. EaD. I. Título.

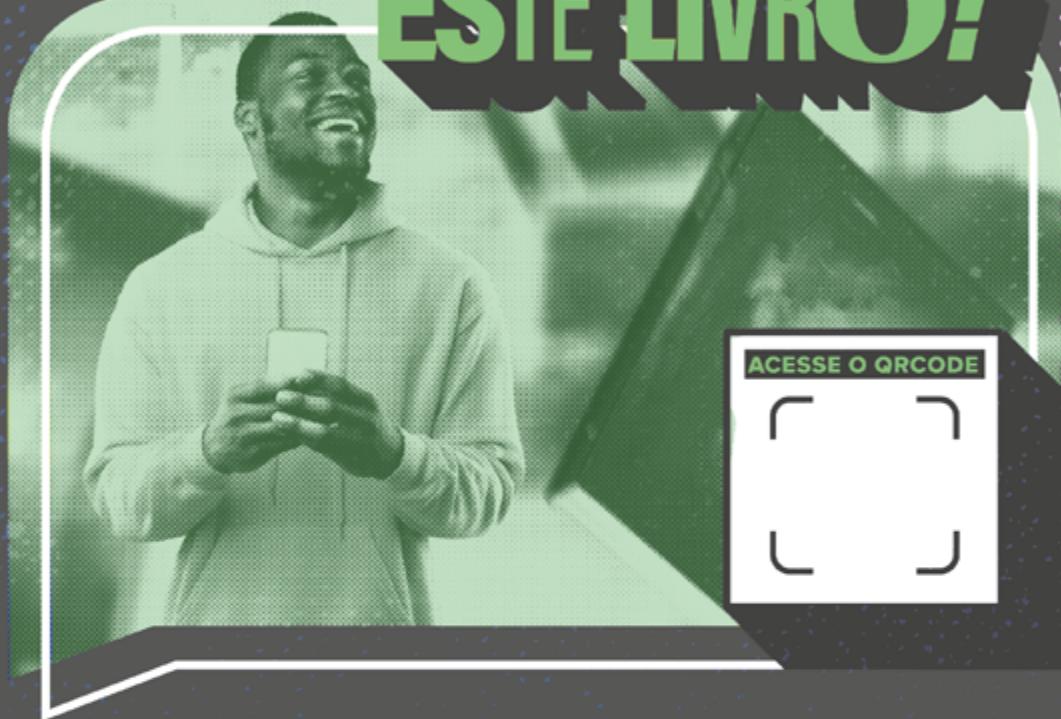
CDD - 005.133

Bibliotecária: Leila Regina do Nascimento - CRB- 9/1722.

Ficha catalográfica elaborada de acordo com os dados fornecidos pelo(a) autor(a).

Impresso por:

AVALIE ESTE LIVRO!



ACESSE O QR CODE



CRIAR MOMENTOS DE APRENDIZAGENS
INESQUECÍVEIS É O NOSSO OBJETIVO E POR ISSO,
GOSTARIAMOS DE SABER COMO FOI SUA EXPERIÊNCIA.

Conta para nós! leva *menos de 2 minutos*. Vamos lá?!

DIGITE O CÓDIGO

173475

Aa

RESPOnda A
PESQUISA

... ?

...



RECURSOS DE IMERSÃO

APROFUNDANDO

Utilizado para temas, assuntos ou conceitos avançados, levando ao aprofundamento do que está sendo trabalhado naquele momento do texto.

PENSANDO JUNTOS

Este item corresponde a uma proposta de reflexão que pode ser apresentada por meio de uma frase, um trecho breve ou uma pergunta.

ZOOM NO CONHECIMENTO

Utilizado para desmistificar pontos que possam gerar confusão sobre o tema. Após o texto trazer a explicação, essa interlocução pode trazer pontos adicionais que contribuam para que o estudante não fique com dúvidas sobre o tema.

EM FOCO

Utilizado para aprofundar o conhecimento em conteúdos relevantes utilizando uma linguagem audiovisual. Disponibilizado por meio de QR-code.

EU INDICO

Utilizado para agregar um conteúdo externo. Utilizando o QR-code você poderá acessar links de vídeos, artigos, sites, etc. Acrescentando muito aprendizado em toda a sua trajetória.



PLAY NO CONHECIMENTO

Professores especialistas e convidados, ampliando as discussões sobre os temas por meio de fantásticos podcasts.



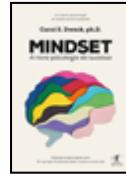
INDICAÇÃO DE FILME

Uma dose extra de conhecimento é sempre bem-vinda. Aqui você terá indicações de filmes que se conectam com o tema do conteúdo.



INDICAÇÃO DE LIVRO

Uma dose extra de conhecimento é sempre bem-vinda. Aqui você terá indicações de livros que agregarão muito na sua vida profissional.



SUMÁRIO

7

UNIDADE 1

PONTEIROS	8
PILHAS E FILAS	38
LISTAS DINÂMICAS	78

113

UNIDADE 2

TÉCNICAS DE ORDENAÇÃO	114
ALGORITMOS DE ORDENAÇÃO AVANÇADOS	140
ÁRVORES	168

191

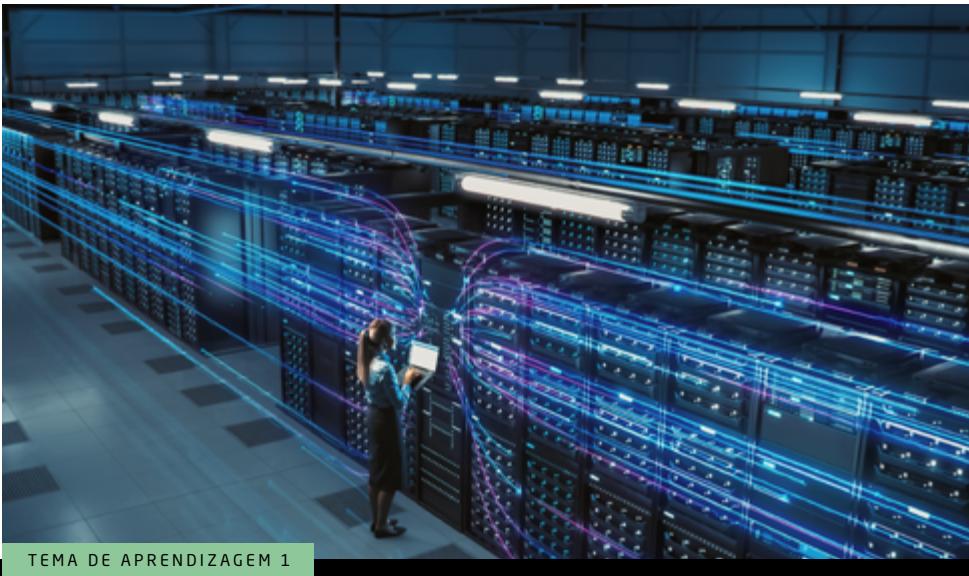
UNIDADE 3

OPERAÇÕES EM ÁRVORES	192
GRAFOS	230
BUSCA EM GRAFOS	260



*uni
dade*

The graphic design features a large, bold, black sans-serif typeface for the word "uni". Below it, the word "dade" is written in a similar font, with the letter "d" being the most prominent. A horizontal yellow bar underlines the "d" and "a". The entire composition is set against a white background and is partially obscured by abstract black and white geometric shapes, including a large trapezoid-like form above and a smaller one below.



TEMA DE APRENDIZAGEM 1

PONTEIROS

MINHAS METAS

- Revisitar a estrutura de vetores e matrizes.
- Criar novos tipos de estruturas usando registros.
- Aprender o conceito de ponteiros.
- Entender como capturar o endereço de uma variável na memória e armazená-la em um ponteiro.
- Estudar a relação entre os ponteiros e as demais variáveis.
- Verificar as propriedades e aplicações de ponteiros.
- Alocar variáveis dinamicamente em tempo de execução

INICIE SUA JORNADA

Uma das bases fundamentais da Computação são as estruturas de dados. Elas são essenciais para armazenar, organizar e manipular informações de maneira eficaz e eficiente. Além disso, elas nos permitem resolver problemas que envolvem ordenação de grandes conjuntos de dados, pesquisa de informações de forma rápida e criação de algoritmos inteligentes. Vamos embarcar nessa jornada?

Para começar, saiba que existe uma lenda que atormenta o sono dos programadores, e o nome dela é **ponteiro**. Neste tema, você verá de forma simples e com muitos exemplos práticos que não há necessidade de perder o sono por causa dos ponteiros.

Essa estrutura é uma das mais importantes, porque graças a ela é possível **fazer alocação dinâmica na memória, navegar nela** para a frente e para trás a partir de uma variável ou de qualquer endereço.

Um ponteiro permite, ainda, que você **monitore endereços na memória, atribua e recupere valores de variáveis** sem ao menos tocá-las. Todavia, antes de adentrar nesse novo ambiente controlado por ponteiros, faremos uma pequena revisão sobre estruturas homogêneas e heterogêneas, pois elas são importantes para o entendimento desse assunto.

Entendendo isso, você passará a se sentir mais confiante e ampliará definitivamente os seus horizontes como programador.



PLAY NO CONHECIMENTO

Que tal conhecer algumas curiosidades sobre os ponteiros e ver como eles são aplicados no mercado de trabalho? Ouça o podcast e mergulhe nesse conceito fascinante que são os ponteiros! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

DESENVOLVA SEU POTENCIAL PONTEIROS

ESTRUTURAS HOMOGÊNEAS E HETEROGLÉNEAS

A primeira estrutura que estudamos foi a variável. Ela é um local reservado na memória para armazenamento de dados. Cada variável pode armazenar apenas uma única informação. Em alguns momentos, porém, é necessário guardar muitas informações e a primeira solução em vista seria declarar variáveis em quantidade suficiente para atender a toda a demanda.

Isso tem muitas desvantagens. Para um programa que vai ler cinco entradas, não é algo muito trabalhoso, mas imagine ter que ler 50, 100 ou 1000 valores, seria necessário criar muitas variáveis, muito código destinado a leitura, processamento e saída.

VETORES E MATRIZES

Para esses casos, a maioria das linguagens de programação traz estruturas prontas para armazenamento múltiplo em uma única variável. Elas são classificadas em homogêneas, que armazenam um único tipo de informação, e heterogêneas, que podem armazenar informações de diferentes tipos. A declaração de um vetor na linguagem C é muito simples, basta declarar uma variável e colocar o seu tamanho entre colchetes logo após o nome. Pense no vetor como uma matriz de uma única linha e quantidade de colunas equivalente ao seu tamanho. O vetor é uma estrutura homogênea, por isso só pode armazenar um único tipo de dado. Exemplo da declaração em linguagem C de um vetor chamado dados com capacidade para armazenar 5 valores inteiros:

```
int dados[5];
```

Na linguagem C, o índice dos vetores e matrizes começa no valor 0 e vai até n-1, em que n é o tamanho do vetor. No exemplo citado, para acessar a primeira posição da variável dados, usa-se o índice 0, e a última, o índice 4, assim:

```
dados[0]; // primeira posição do vetor dados  
dados[1]; // segunda posição  
dados[2];  
dados[3];  
dados[4]; // quinta e última posição
```

As matrizes possuem pelo menos duas dimensões. A declaração é parecida com a de vetores, precisando indicar também a quantidade de linhas além da quantidade de colunas. A seguir, veja um exemplo de declaração de uma matriz de números reais, com duas linhas e três colunas:

```
float matriz[2][3];
```

Lembre-se de que são necessários dois índices para acessar os dados em uma matriz bidimensional, conforme os exemplos a seguir:

```
matriz[0][0]; // primeira linha, primeira coluna  
matriz[0][1]; // primeira linha, segunda coluna  
matriz[1][2]; // segunda e última linha, terceira e última  
coluna
```

REGISTROS

O registro é uma coleção de variáveis e, por ser uma estrutura heterogênea, permite o armazenamento de informações de diferentes tipos. Ele possibilita que o programador crie tipos de dados específicos e personalizados. A declaração de um registro se dá pela palavra reservada **struct**, seguida pelo conjunto de elementos que o compõem. Veja um exemplo de um registro chamado fraction, que possui três elementos: numerator, denominator e value:

```
struct fraction {
    int numerator;
    int denominator;
    float value;
}
```

Após declarado o registro, o seu uso se dá como tipo de variável, assim como usado para inteiros, reais, caracteres etc. Cada elemento do registro é acessado por meio de uma referência composta pelo **nome_da_variável.nome_do_elemento**, conforme exemplos a seguir:

```
struct fraction metade; // cria uma variável do tipo fraction
metade.numerator = 1; // atribui valor ao elemento numerator

metade.denominator = 2; // atribui valor ao elemento denominator

metade.value = metade.numerator / metade.denominator
```

É possível criar vetores e matrizes para acomodar múltiplos registros. Vamos definir um registro chamado livro para armazenar quatro notas e depois vamos criar um vetor para armazenar as notas de 40 alunos:

```
struct livro {
    float nota1;
    float nota2;
    float nota3;
    float nota4;
}
struct livro alunos_notas[40];
```

PONTEIROS

Uma variável é um objeto que representa um espaço reservado na memória. Quando escolhemos o tipo da variável, estamos definindo o tamanho de bytes que ela terá e as regras de como seus bits serão lidos, conforme foi discutido no início deste tema.

Um inteiro tem 4 bytes (32 bits), assim como um número real, só que no número inteiro positivo todos os bits são significativos, enquanto na variável de ponto flutuante só os primeiros 24 representam valor, os últimos 8 determinam a posição da casa decimal no número.

Por isso, quando encontramos uma variável de 4 bytes alocada na memória, precisamos saber qual o seu tipo para fazer a sua correta leitura.

Em vez de obter o valor armazenado numa variável, podemos opcionalmente obter o seu endereço na memória. Por exemplo, criamos uma variável *x* do tipo inteiro; para saber qual o seu endereço, usamos a notação *&x*. Isso significa que *&x* é um ponteiro que aponta para o endereço da variável *x* na memória.

Também é possível usar um ponteiro como tipo de dado na declaração de uma variável, só que nesse caso ele não irá guardar um valor, mas sim uma posição na memória. Vejamos agora exemplos de criação de variáveis e ponteiros:

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

int xi;
int *ptr_xi;

float xf;
float *ptr_xf;

char xc;
char *ptr_xc;

int main() {
    system("Pause");
    return(0);
}
```

A variável *xi* é do tipo inteiro e *ptr_xi* é um ponteiro que aponta para uma variável do tipo inteiro. A mesma relação existe para *xf* e *ptr_xf*, só que no caso deles é para armazenar um valor de ponto flutuante e um ponteiro para uma variável do tipo ponto flutuante. Por último, *xc* é uma variável do tipo caractere e *ptr_xc*, um ponteiro para um caractere.

Segundo Tenenbaum (1995, p. 29),



[...] um ponteiro é como qualquer outro tipo de dado em C. O valor de um ponteiro é uma posição na memória da mesma forma que o valor de um inteiro é um número. Os valores dos ponteiros podem ser atribuídos como quaisquer outros valores.

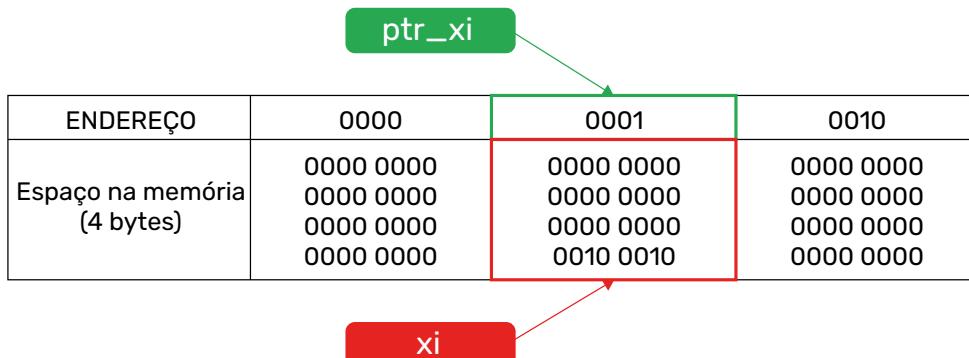


Figura 1 - Simulação de espaço na memória / Fonte: a autora.

Descrição da Imagem: a figura contém uma tabela composta por duas linhas e quatro colunas. Na primeira coluna, temos os títulos de cada linha e nas demais colunas temos o conteúdo. A primeira linha da tabela contém endereços de memória, enquanto a segunda linha contém o conteúdo em si, sendo estes armazenados em espaços de memória de 4 bytes. O endereço de memória é representado pelo ponteiro `ptr_xi` e o conteúdo pela variável `xi`. No primeiro endereço, temos como valor do ponteiro 0000 e seu conteúdo (`xi`) é 0000 0000 0000 0000 0000 0000 0000 0000. No segundo endereço, temos como valor do ponteiro 0001 e seu conteúdo (`xi`) é 0000 0000 0000 0000 0000 0000 0000 0010. No terceiro endereço, temos como valor do ponteiro 0010 e seu conteúdo (`xi`) é 0000 0000 0000 0000 0000 0000 0000 0000. Em volta do segundo endereço de memória (0001), temos um contorno na cor verde e uma indicação de que se trata do ponteiro `ptr_xi` e em volta do segundo conteúdo temos um contorno vermelho e uma indicação de que se trata do conteúdo armazenado em `xi`. Fim da descrição.

Como `ptr_xi` é um ponteiro, não posso simplesmente atribuir a ele o valor de `xi`, preciso, sim, atribuir o endereço que `xi` ocupa na memória. Para isso, usamos a notação `&xi`, que significa o ponteiro que aponta para o endereço na memória da variável `xi`.

```
ptr_xi = &xi;
```

Sabe-se que ptr_xi contém o endereço de uma variável, mas como saber o valor daquele objeto? Para isso, é usada a notação $*ptr_xi$, que significa: o valor da variável para qual aponta o ponteiro ptr_xi .

```
xi = *ptr_xi;
```

Alterei a imagem anterior e inclui as duas novas notações ($\&xi$ e $*ptr_xi$) para demonstrar melhor as suas relações (Figura 2).

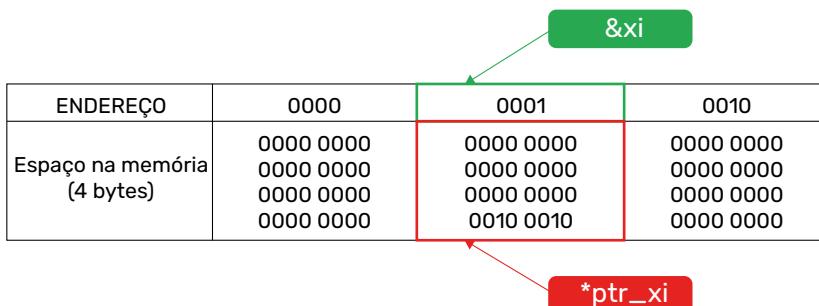


Figura 2 - Simulação de espaço na memória usando as notações & e * / Fonte: a autora.

Descrição da Imagem: a figura contém uma tabela composta por duas linhas e quatro colunas. Na primeira coluna, temos os títulos de cada linha e nas demais colunas temos o conteúdo. A primeira linha da tabela contém endereços de memória, enquanto a segunda linha contém o conteúdo em si, sendo estes armazenados em espaços de memória de 4 bytes. O endereço de memória é representado pelo ponteiro ptr_xi e o conteúdo pela variável xi . No primeiro endereço, temos como valor do ponteiro 0000 e seu conteúdo (xi) é 0000 0000 0000 0000 0000 0000 0000 0000. No segundo endereço, temos como valor do ponteiro 0001 e seu conteúdo (xi) é 0000 0000 0000 0000 0000 0000 0000 0010. No terceiro endereço, temos como valor do ponteiro 0010 e seu conteúdo (xi) é 0000 0000 0000 0000 0000 0000 0000 0000. Em volta do segundo endereço de memória (0001), temos um contorno na cor verde e uma indicação de que se trata do endereço de xi , indicado pela notação $\&xi$ e, em volta do segundo conteúdo, temos um contorno vermelho e uma indicação de que se trata do valor da variável para qual aponta o ponteiro ptr_xi , indicado pela notação $*ptr_xi$. Fim da descrição.

Existem, ainda, outros conceitos interessantes sobre ponteiros, porém é necessário primeiramente fixar o que foi visto até aqui antes de seguirmos com o conteúdo. Vamos partir para a prática!

Primeiramente, vamos criar duas variáveis. A primeira será xi do tipo inteiro e a segunda será ptr_xi do tipo ponteiro de inteiro.

```
int xi;
int *ptr_xi;
```

Agora vamos fazer uma função chamada `imprimir()`, que vai desenhar na tela o valor de xi , $\&xi$, ptr_xi e $*ptr_xi$.

```
void imprimir() {
    printf("Valor de xi = %d \n", xi);
    printf("Valor de &xi = %p \n", &xi);
    printf("Valor de ptr_xi = %p \n", ptr_xi);
    printf("Valor de *ptr_xi = %d \n\n", *ptr_xi);

}
```

Lembrando que xi é uma variável do tipo inteiro e $\&xi$ é o ponteiro que aponta para o endereço onde xi está armazenada na memória. A variável ptr_xi é um ponteiro para um inteiro e $*ptr_xi$ é o valor para o qual o ponteiro ptr_xi está apontando.

Dentro da função `main()`, vamos atribuir o valor 10 para xi e o valor de $\&xi$ para ptr_xi . Em seguida, vamos chamar a função `imprimir()` e observar o resultado.

```
int main() {

    xi = 10;
    ptr_xi = &xi;
    imprimir();

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

A primeira coisa que a função `imprimir()` faz é mostrar o valor de xi , que sabemos ser 10. Em seguida, imprime o endereço de memória de xi que é obtido pela notação $\&xi$. A próxima saída é o valor de ptr_xi , que agora aponta para o endereço da variável xi , e, por último, o valor de $*ptr_xi$, que é o conteúdo para onde ptr_xi está apontando.

```
Valor de xi = 10
Valor de &xi = 00405020
Valor de ptr_xi = 00405020
Valor de *ptr_xi = 10
```

Note que o valor de *ptr_xi* é o mesmo que *&xi*, posto que, quando usamos a notação *&xi*, conseguimos o endereço de memória da variável *xi* e o ponteiro *ptr_xi* está apontando exatamente para ele. Quando usamos a notação **ptr_xi*, conseguimos acessar o endereço de *xi* e resgatar o seu valor armazenado.

Vamos fazer algo diferente agora. Após as atribuições iniciais, antes de chamar a função *imprimir()*, vamos alterar o valor da variável *xi* para 20.

```
int main() {
    xi = 10;
    ptr_xi = &xi;
    xi = 20;
    imprimir();

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

O que devemos esperar como saída? Dá para dizer, sem titubear, que o valor de *xi* será 20, e não 10, mas e o resto das variáveis e notações, o que elas irão nos revelar?

```
Valor de xi = 20
Valor de &xi = 00405020
Valor de ptr_xi = 00405020
Valor de *ptr_xi = 20
```

Tanto *ptr_xi* quanto *&xi* mantêm o mesmo valor, visto que não houve alteração da posição na memória que ocupa a variável *xi*. Apesar de termos alterado apenas o valor de *xi*, porém o valor de **ptr_xi* também aparece diferente. Como isso é possível? Como o ponteiro *ptr_xi* aponta para a variável *xi*, qualquer alteração feita em seu conteúdo irá refletir automaticamente quando verificamos o valor de **ptr_xi*.

Vamos fazer mais uma alteração agora, aproveitando tudo o que já foi feito? Só que, em vez de alterar o valor de xi , vamos tentar alterar o valor de $*ptr_xi$ para 30.

```
int main() {
    xi = 10;
    ptr_xi = &xi;
    xi = 20;
    *ptr_xi = 30;
    imprimir();

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

Já sabemos que o valor de xi não será 10, já que o atualizamos para 20 antes da impressão, não é? Fizemos apenas uma alteração no valor de $*ptr_xi$ e sabemos que o ponteiro ptr_xi aponta para o conteúdo de xi , não o contrário. Repare no que aconteceu agora:

```
Valor de xi = 30
Valor de &xi = 00405020
Valor de ptr_xi = 00405020
Valor de *ptr_xi = 30
```

Por que o valor de xi foi alterado quando modificamos o valor de $*ptr_xi$? Essa é fácil de responder. O ponteiro ptr_xi aponta para o local onde xi está armazenado, ou seja, o endereço $\&xi$. Quando atribuímos 30 para $*ptr_xi$, não alteramos o valor do ponteiro ptr_xi , mas sim o valor da variável que o ponteiro estava apontando, que é xi .

Acabamos de colocar em prática diversos conceitos interessantes. Foi possível entender como atribuir a um ponteiro o endereço de uma variável. Vimos também que, quando alteramos o valor de uma variável, esse novo valor reflete

no conteúdo para o qual o ponteiro está apontando. Por fim, vimos que é possível alterar o valor de uma variável sem fazer uma atribuição direta a ela, apenas manipulando um ponteiro que aponta para o seu endereço.

A seguir, é apresentado o código-fonte completo para o exemplo que acabamos de criar. Experimente digitá-lo no seu compilador e executá-lo. Compare as saídas que você obteve com as demonstradas aqui. Faça algumas alterações nas atribuições e nos valores e observe o resultado.

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

int xi;
int *ptr_xi;

void imprimir() {
    printf("Valor de xi = %d \n", xi);
    printf("Valor de &xi = %p \n", &xi);
    printf("Valor de ptr_xi = %p \n", ptr_xi);
    printf("Valor de *ptr_xi = %d \n\n", *ptr_xi);
}

int main() {
    xi = 10;
    ptr_xi = &xi;
    imprimir();

    xi = 20;
    imprimir();

    *ptr_xi = 30;
    imprimir();

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

Essas são as saídas esperadas para o algoritmo descrito:

```

Valor de xi = 10
Valor de &xi = 00405020
Valor de ptr_xi = 00405020
Valor de *ptr_xi = 10
Valor de xi = 20
Valor de &xi = 00405020
Valor de ptr_xi = 00405020
Valor de *ptr_xi = 20
Valor de xi = 30
Valor de &xi = 00405020
Valor de ptr_xi = 00405020
Valor de *ptr_xi = 30

```

PROPRIEDADES DE PONTEIROS

É importante ressaltar que, quando criamos um ponteiro, não apenas criamos uma variável que aponta para um endereço, mas também uma variável que aponta para um endereço de um determinado tipo.

Como vimos no início do tema, cada tipo de dado possui as suas regras e organização lógica. Se olharmos o conteúdo de um bloco de 4 bytes, o mesmo conteúdo pode ser interpretado de formas diferentes, caso seja um inteiro ou um ponto flutuante.

Dessa forma, quando criamos *ptr_xi*, criamos um ponteiro que aponta para uma variável do tipo inteiro, assim como *ptr_xf* aponta para uma variável de tipo flutuante e *ptr_xc* aponta para um caractere.

```

int xi;
int *ptr_xi;

float xf;
float *ptr_xf;

char xc;
char *ptr_xc;

```

Contudo, é possível em C fazer a conversão de ponteiros de tipos diferentes. Por exemplo, podemos converter o valor de *ptr_xf* para o tipo ponteiro para um inteiro por meio do comando (*int **).

```
ptr_xi = (int *) ptr_xf;
```



PENSANDO JUNTOS

Pare tudo! Pare o mundo! Agora você pode se perguntar: "mas por que tudo isso? Esses detalhes são realmente necessários?". Não é uma questão de puro preciosismo dos criadores da linguagem, eu vou explicar agora o porquê de tudo isso.

O ponteiro é um tipo de dado, e eu posso criar uma variável com esse tipo, certo? E é possível aplicar diversos operadores aritméticos em variáveis, não é verdade? Se *ptr_xi* contém o endereço de um valor inteiro na memória, *ptr_xi+1* irá apontar para o endereço do número inteiro imediatamente posterior a **ptr_xi*, e *ptr_xi - 1* irá apontar para o primeiro endereço de um número inteiro imediatamente anterior a **ptr_xi*. Dessa forma, a partir de um ponteiro, é possível navegar pela memória investigando os valores de outros endereços sem precisar saber a qual variável eles pertencem (Figura 3).

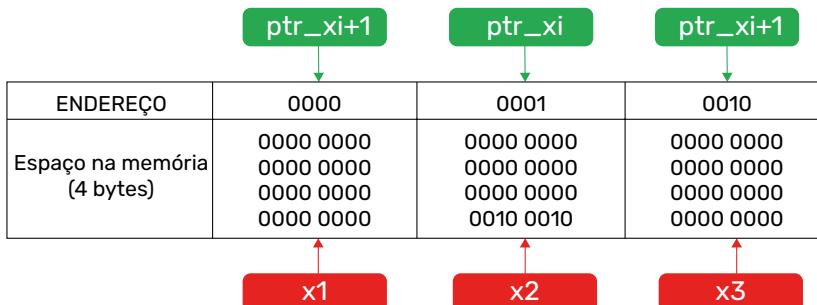


Figura 3 - Navegação pela memória por meio de ponteiros / Fonte: o organizador.

Descrição da Imagem: a figura contém uma tabela composta por duas linhas e quatro colunas. Na primeira coluna, temos os títulos de cada linha e nas demais colunas temos o conteúdo. A primeira linha da tabela contém endereços de memória, enquanto a segunda linha contém o conteúdo em si, sendo estes armazenados em espaços de memória de 4 bytes. O endereço de memória é representado pelo ponteiro *ptr_xi* e o conteúdo pela variável *xi*. No segundo endereço, temos como valor do ponteiro 0001, representado por *ptr_xi* e seu conteúdo (*xi*) é 0000 0000 0000 0000 0000 0000 0010 0010. Conclui-se que o primeiro endereço é representado por *ptr_xi-1*, sendo o valor armazenado em *xi* igual a 0000 0000 0000 0000 0000 0000 0000. Por fim, no terceiro endereço, temos como valor do ponteiro 0010, sendo este representado por *ptr_xi+1* e seu conteúdo (*xi*) é 0000 0000 0000 0000 0000 0000 0000 0000. Fim da descrição.

Suponha que o nosso hardware enderece cada posição na memória byte a byte e o compilador use 4 bytes para cada variável do tipo inteiro. Se *ptr_xi* está apontando para o valor armazenado na posição 100 da memória, quando procurarmos *ptr_xi + 1*, estaremos acessando o endereço 104, posto que a variável atual começa na posição 100, mas um valor inteiro ocupa 4 bytes, ou seja, da posição 100 à posição 103. Analogamente, quando fizermos *ptr_xi - 1*, estaremos acessando a posição 96.

Como definimos que o ponteiro aponta para um tipo inteiro, ele sabe que cada variável possui 4 bytes. Dessa forma, quando usamos a notação $*(ptr_xi + 1)$, estaremos acessando o valor inteiro formado pelos bytes 104, 105, 106 e 107, e para $*(ptr_xi - 1)$ o inteiro formado pelos bytes 96, 97, 98 e 99.

ALOCAÇÃO DINÂMICA NA MEMÓRIA

Até o momento, fizemos apenas alocação estática de objetos. Definimos variáveis dentro do corpo do código-fonte e, quando o programa é executado, todas as variáveis são criadas e inicializadas na memória.

Existe, porém, uma forma de alocar variáveis dinamicamente, criando mais e mais espaços de acordo com a necessidade do programador, sem haver necessidade de prevê-las durante o desenvolvimento do programa. Observe esse pequeno trecho de código:

```
int *ptr;  
printf ("Endereço: %p\n\n", ptr);
```

Definimos uma variável do tipo ponteiro de inteiro chamado *ptr*. Como toda variável, ela é inicializada pelo compilador durante a carga do programa. Uma variável numérica (inteiros, pontos flutuantes) tem o seu valor inicializado com 0, já um ponteiro é inicializado com um endereço inválido na memória, ou seja, ele não aponta para local algum. Não é possível obter o valor de **ptr*, pois se tentarmos ver o seu conteúdo, o programa encerrará com um erro. No entanto, podemos observar para onde ele aponta nesse exato momento.

```
Endereco: 00000008
```

Por enquanto, não temos nenhum interesse em criar uma nova variável, mas, então, onde iremos usar o ponteiro *ptr*? Todo ponteiro aponta para um endereço na memória e, nos exemplos anteriores, usamos os ponteiros para guardar o endereço de outras variáveis. Vamos adicionar agora uma linha no nosso programa para fazer a alocação dinâmica de um espaço na memória.

```
int *ptr;
printf ("Endereco: %p\n\n", ptr);
ptr = (int *) malloc(sizeof (int));
printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);
```

A função *malloc* reserva um espaço na memória do tamanho *sizeof*. No caso de um inteiro, sabemos que C o considera com 4 bytes. Como *ptr* é um ponteiro do tipo inteiro, precisamos nos certificar de que o retorno de *malloc* seja um valor compatível e, por isso, o uso de (*int **). O retorno dessa execução pode ser visto a seguir:

```
Endereco: 00000008
Endereco: 00380CA0
Valor: 3673280
```

Note que, após a alocação na memória pela função *malloc*, o ponteiro *ptr* já passa a apontar para um endereço válido disponibilizado pelo compilador. No entanto, quando um espaço na memória é reservado de forma dinâmica, seu valor não é inicializado tal qual na declaração de uma variável estática.

Observe que o valor de **ptr* é 3673280, mas de onde surgiu esse valor? O compilador buscou na memória um bloco de 4 bytes disponível e passou esse endereço para *ptr*, porém aqueles bytes já possuíam valores. A memória do computador só é zerada quando é desligado e sua fonte de alimentação de energia interrompida, ou ainda por algum comando específico do sistema operacional. As variáveis são criadas e destruídas na memória, mas os valores dos bits continuam inalterados.

Por isso, é sempre importante inicializar o valor de uma variável criada de forma dinâmica. Acompanhe o novo trecho do programa:

```
int *ptr;  
printf ("Endereco: %p\n\n", ptr);  
ptr = (int *) malloc(sizeof (int));  
printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);  
*ptr = 10;  
printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);
```

Agora que *ptr* está apontando para um endereço válido na memória, podemos sem problema algum ir até lá e armazenar um valor em **ptr*. Note que após a atribuição de **ptr = 10* o endereço do ponteiro continua inalterado.

```
Endereco: 00000008  
Endereco: 00380CA0  
Valor: 3673280  
Endereco: 00380CA0  
Valor: 10
```

ZOOM NO CONHECIMENTO

Um ponteiro aponta para um endereço na memória, seja ele de uma variável pre-definida ou alocada dinamicamente. Tome cuidado para não passar um endereço diretamente para um ponteiro, deixe esse trabalho para o compilador. Você pode acabar acessando uma porção da memória ocupada por outro programa ou pelo próprio sistema operacional.

Nós já declaramos *ptr* e alocamos um espaço na memória para ele. Vamos criar agora uma variável *x* do tipo inteira, inicializá-la com valor 20 e atribuir o endereço *&x* para *ptr*.

```
int x;
x = 20;
ptr = &x;
printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);
```

O que podemos esperar na saída de *printf*? Acertou se disse que *ptr* agora aponta para um novo endereço na memória, o endereço da variável *x*. Como *ptr* aponta para *&x*, o valor de **ptr* passa a retornar o valor de *x*, que é 20.

```
Endereco: 0028FF08
Valor: 20
```

E o que aconteceu com o endereço original de *ptr*? Ele se perdeu, já que não era de nenhuma variável, mas alocado dinamicamente para o ponteiro. Vamos, então, fazer um novo *malloc* para *ptr* e ver qual o resultado.

```
int x;
x = 20;
ptr = &x;
printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);

ptr = (int *) malloc(sizeof (int));
printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);
```

Veremos que *ptr* agora aponta para um novo endereço na memória, endereço esse diferente do primeiro obtido pela função *malloc* e diferente do endereço da variável inteira *x*.

```
Endereco: 0028FF08
Valor: 20

Endereco: 00380CC0
Valor: 3683664
```

Novamente, `*ptr` traz o “lixo” de bits existente no endereço que acabou de ser alocado a ele pela função `malloc`. Essa função traz o endereço de um espaço disponível na memória, mas não faz nenhum tratamento com a atual string de bits naquela posição. A seguir, você encontra o código-fonte completo do exemplo que acabamos de estudar. Leia com atenção e tente entender cada uma das linhas do programa.

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

main() {
    int *ptr;
    printf ("Endereco: %p\n\n", ptr);
    ptr = (int *) malloc(sizeof (int));
    printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);
    *ptr = 10;
    printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);

    int x;
    x = 20;
    ptr = &x;
    printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);

    ptr = (int *) malloc(sizeof (int));
    printf ("Endereco: %p \nValor: %d\n\n", ptr, *ptr);

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

Os endereços que aparecem, ao executarmos o exemplo, variam cada vez que o programa é inicializado, porque o compilador sempre procura novas posições livres na memória no momento de criação de variáveis estáticas e dinâmicas. Execute o código algumas vezes e observe como os endereços variam.

```
Endereco: 00000008
```

```
Endereco: 00380CA0
```

```
Valor: 3673280
```

```
Endereco: 00380CA0
```

```
Valor: 10
```

```
Endereco: 0028FF08
```

```
Valor: 20
```

```
Endereco: 00380CC0
```

```
Valor: 3683664
```

CRIANDO VETORES DINÂMICOS

Quando criamos um vetor, dizemos na sua declaração qual será o seu tamanho. Independentemente se vamos usá-lo por completo ou uma parte, todo o espaço na memória é reservado assim que o programa é inicializado.

Isso é ruim por dois motivos:

1. Pode-se reservar mais espaço do que realmente precisamos.
2. Ficamos limitados à quantidade de posições previamente definidas no vetor.

Uma forma de resolver esses problemas é criar vetores dinâmicos, com seu tamanho definido em tempo de execução do programa. Vamos criar um pequeno exemplo para ilustrar isso.

Na função *main()*, vamos criar uma variável *tam* do tipo inteiro para ler o tamanho do vetor e uma variável do tipo ponteiro de inteiro chamada *vetor*, que será criada de forma dinâmica de acordo com o valor lido na variável *tam*.

```
int tam;
int *vetor;
printf ("Escolha o tamanho do vetor: ");
scanf("%d", &tam);
vetor = (int *) malloc(sizeof (int)*tam);
```

O processo de alocação de um vetor usando *malloc* é muito parecido com o de uma variável simples. Basta multiplicar o tipo definido em *sizeof* pela quantidade de posições que deseja para o vetor.

Para comprovar o funcionamento desse exemplo, vamos fazer um laço de repetição e colocar em cada posição do novo vetor uma potência de base 2 elevada à sua posição definida pelo índice *i*.

```
for (int i = 0; i < tam; i++) {  
    vetor[i] = pow(2,i);  
    printf ("Posicao %d: %d\n", i, vetor[i]);  
}
```

Esse programa será muito útil quando for resolver exercícios em que se precisa saber o valor de várias potências de 2, como no caso da conversão de números binários em decimais. A execução do exemplo com *tam* = 5 será:

```
Escolha o tamanho do vetor: 5  
Posicao 0: 1  
Posicao 1: 2  
Posicao 2: 4  
Posicao 3: 8  
Posicao 4: 16
```

Analise o código completo do exemplo, copie no compilador C de sua preferência e execute o programa lendo diferentes valores para o tamanho *tam*.

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

int main() {
    int tam;
    int *vetor;
    printf ("Escolha o tamanho do vetor: ");
    scanf ("%d", &tam);
    vetor = (int *) malloc(sizeof (int)*tam);
    for (int i = 0; i < tam; i++) {
        vetor[i] = pow(2,i);
    }
    printf ("Posicao %d: %d\n", i, vetor[i]);
    system("Pause");
    return(0);
}
```



INDICAÇÃO DE LIVRO

Projeto de algoritmos.

Algoritmos e estruturas de dados formam o núcleo da ciência da computação, sendo os componentes básicos de qualquer software. Aprender algoritmos é crucial para qualquer pessoa que deseja desenvolver um software de qualidade. Esta obra apresenta os principais algoritmos conhecidos.

Destacam-se as técnicas de projeto de algoritmos ensinadas por meio de refinamentos sucessivos até o nível de um programa em Pascal. No livro, todo programa Pascal tem um programa C correspondente no apêndice. Além disso, o livro possui mais de 163 exercícios propostos, dos quais 80 com soluções; 221 programas em Pascal e 221 programas em C; e 164 figuras ilustrativas.



EM FOCO

Assista às aulas para complementar o conteúdo deste tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

NOVOS DESAFIOS

Até aqui, você programou de forma estática. Ficou limitado a criar variáveis somente durante a codificação do sistema. Encontrava situações em que a variável ocupava mais espaço na memória do que realmente precisaria, ou o contrário, com espaço insuficiente para tanta informação.

Isso tudo mudou com o conteúdo visto neste tema. O ponteiro é uma estrutura de dados muito poderosa e permite, além de outras coisas, fazer alocações dinâmicas na memória.

Ponteiros são importantes quando trabalhamos com algoritmos e estruturas de dados. Listas encadeadas, árvores e gráficos, frequentemente, fazem uso de ponteiros para organizar e acessar os dados de forma eficiente. Por isso, conhecer e saber implementar códigos usando ponteiros são pontos importantes para nós, pessoas desenvolvedoras.



AUTOATIVIDADE

1. A alocação dinâmica de memória, realizada através da função malloc, possibilita a criação de blocos de memória de tamanho variável em tempo de execução, proporcionando maior eficiência e otimização de recursos. O uso adequado de ponteiros e alocação dinâmica requer atenção e cuidado, mas, quando empregados corretamente, eles conferem ao programador maior controle e flexibilidade na manipulação de memória e no desenvolvimento de estruturas de dados complexas.

Fonte: STROUSTRUP, B. **The C++ Programming Language**. 4th ed. Addison-Wesley Professional, 2013.

- Faça um pequeno programa para testar seus conhecimentos sobre ponteiros e alocação dinâmica de memória.
 - Defina um ponteiro do tipo inteiro.
 - Faça alocação dinâmica para o ponteiro recém-criado.
 - Preencha a memória com o valor 42.
 - Imprima o endereço do ponteiro na memória e o valor contido nele.
2. Ao alocar dinamicamente na memória durante a execução de um programa, os desenvolvedores podem criar vetores, listas encadeadas, matrizes e outras estruturas cujo tamanho é determinado em tempo de execução. Isso evita o desperdício de memória, pois os recursos são alocados apenas quando necessário.

Fonte: KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language**. 2nd ed. Prentice Hall. 1988.

- Crie um programa que leia uma variável e crie dois vetores dinâmicos, um com o tamanho lido e outro com o dobro desse tamanho. Preencha esses vetores, respectivamente, com potências de 2 e potências de 3.
- Crie uma variável inteira e dois ponteiros do tipo inteiro.
- Pergunte ao usuário o tamanho do vetor dinâmico e leia esse valor na variável inteira.
- Aloque dinamicamente os dois vetores usando a função malloc.
- Faça um laço de repetição para preencher o primeiro vetor com potências de 2.
- Faça um segundo laço de repetição para preencher o outro vetor com potências de 3.
- Não se esqueça de usar a biblioteca math.h para poder usar o cálculo de potência (pow).

AUTOATIVIDADE

3. Ao utilizar ponteiros, os programadores podem implementar estruturas de dados e algoritmos mais complexos. No entanto, o uso adequado de ponteiros requer atenção e cuidado para evitar problemas de segurança, como vazamento de memória ou falhas de segmentação.

Fonte: STROUSTRUP, B. **The C++ Programming Language**. 4th ed. Addison-Wesley Professional, 2013.

Em linguagens de programação de baixo nível, como C, qual das seguintes alternativas descreve corretamente o conceito de ponteiros?

- a) Os ponteiros são tipos de dados utilizados para armazenar valores numéricos inteiros.
 - b) Os ponteiros são variáveis que armazenam endereços de memória de outras variáveis.
 - c) Os ponteiros são estruturas de dados utilizadas para armazenar sequências de elementos do mesmo tipo.
 - d) Os ponteiros são utilizados para declarar funções com um grande número de parâmetros.
 - e) Os ponteiros são tipos de dados utilizados para armazenar valores literais.
4. O operador de desreferência (*) é utilizado para acessar o valor armazenado no endereço de memória apontado pelo ponteiro. Essa capacidade é essencial para manipulação eficiente de dados, especialmente quando se deseja modificar o valor da variável original através do ponteiro. Ao utilizar o operador de desreferência, os programadores conseguem acessar e modificar indiretamente os dados na memória, tornando o trabalho com ponteiros uma ferramenta poderosa e versátil

Fonte: STROUSTRUP, B. **The C++ Programming Language**. 4th Edition. Addison-Wesley Professional, 2013.

Analise o código a seguir e indique a alternativa correta sobre o que acontecerá ao ser executado:

```
#include <stdio.h>
int main() {
    int xi = 20;
    int *ptr;
    ptr = &xi;
    printf("Endereço de memória de xi: %p\n", &xi);
    printf("Conteúdo apontado por ptr: %d\n", *ptr);
    return 0;
}
```

AUTOATIVIDADE

- a) O código irá resultar em um erro de compilação devido ao uso inadequado de ponteiros.
 - b) Será impresso o endereço de memória da variável *xi* e o valor 20 apontado pelo ponteiro *ptr*.
 - c) O código não irá compilar, pois falta declarar o tipo de dado do ponteiro *ptr*.
 - d) Será exibido o valor do ponteiro *ptr*, e não o valor da variável *xi*.
 - e) Será exibido o endereço de memória de *xi* na segunda impressão.
5. O operador & é uma característica fundamental presente em linguagens de programação de baixo nível. Essa funcionalidade é crucial para a manipulação direta da memória e para operações que envolvem passagem de parâmetros por referência em funções.

Fonte: STROUSTRUP, B. **The C++ Programming Language**. 4th ed. Addison-Wesley Professional, 2013.

Com base nas informações apresentadas, avalie as asserções a seguir e a relação proposta entre elas:

I - O operador & é utilizado para obter o endereço de memória de uma variável.

PORQUE

II - Ao aplicar o operador & a uma variável, o compilador retorna o endereço de memória associado àquela variável, possibilitando seu armazenamento em ponteiros ou seu uso para alocação dinâmica de memória.

A respeito dessas asserções, assinale a opção correta:

- a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
- b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
- c) A asserção I é uma proposição verdadeira e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa e a II é uma proposição verdadeira.
- e) As asserções I e II são falsas.

REFERÊNCIAS

TENENBAUM, A. M. **Estruturas de dados usando C**. Tradução de Teresa Cristina Félix de Souza. São Paulo: MAKRON Books, 1995.

GABARITO

1.

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

int main() {
    int *ptr;
    ptr = (int *) malloc(sizeof (int));
    *ptr = 42;
    printf ("Endereco: %p\nValor: %d\n\n", ptr, *ptr);

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

2.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int tamanho;
int *vetor1, *vetor2;
int main() {
    int i;
    printf ("Escolha o tamanho do vetor: ");
    scanf("%d", &tamanho);
    vetor1 = (int *) malloc(sizeof (int)*tamanho);
    vetor2 = (int *) malloc(sizeof (int)*(tamanho*2));
    printf ("\nVetor1: \n");
    for (i = 0; i < tamanho; i++) {
        vetor1[i] = pow(2,i);
    }
    printf ("Posicao %d: %d\n", i, vetor1[i]);
    printf ("\nVetor2: \n");
    for (i = 0; i < tamanho*2; i++) {
        vetor2[i] = pow(3,i);
    }
    printf ("Posicao %d: %d\n", i, vetor2[i]);
}
system("Pause");
return(0);
}
```

GABARITO

3. **Opção B.** Os ponteiros permitem que os programadores acessem e manipulem diretamente a localização de memória onde uma variável está armazenada.

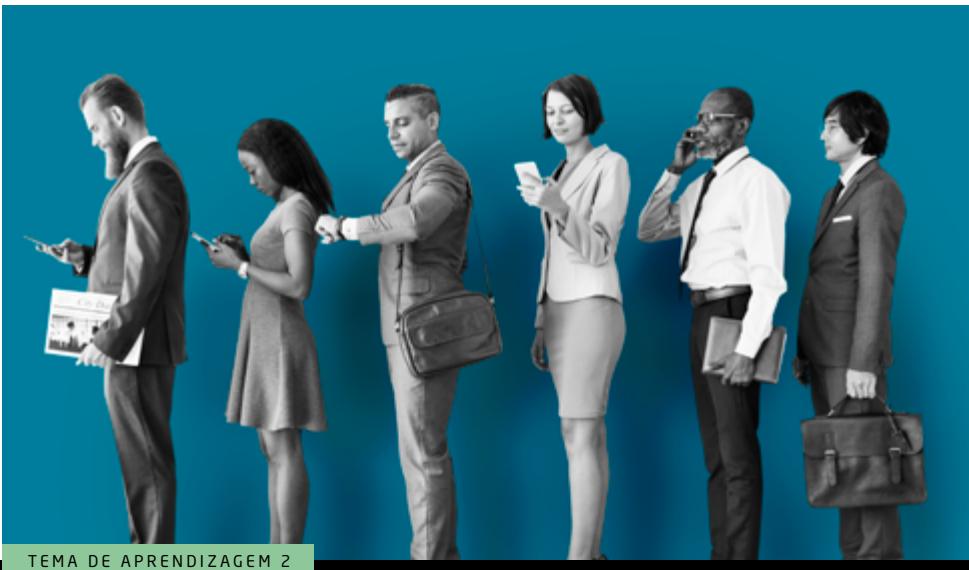
Isso permite que o programador acesse e modifique o valor da variável apontada indiretamente através do endereço de memória armazenado no ponteiro. Conclui-se, então, que os ponteiros não são utilizados para armazenar valores numéricos inteiros, nem literais, nem sequências de elementos do mesmo tipo e não são usados para declarar funções com um grande número de parâmetros, visto que seu principal objetivo é armazenar endereços de memória de outras variáveis.

4. **Opção B.** O código inicializa uma variável *xi* com o valor 20. Em seguida, declara um ponteiro *ptr* do tipo inteiro e atribui a ele o endereço de memória da variável *xi* utilizando o operador &. Ao imprimir o endereço de memória de *xi* com "%p" e o valor apontado por *ptr* (conteúdo da posição de memória apontada pelo ponteiro) com "%d", o programa exibirá a seguinte saída:

Endereço de memória de *xi*: [endereço de memória de *xi*]
Conteúdo apontado por *ptr*: 10

5. **Opção A.** Ao usar o operador & para obter o endereço de memória de uma variável, os programadores têm maior controle e flexibilidade no gerenciamento de memória. Isso torna o operador '&' uma ferramenta essencial para otimização de desempenho e implementação de estruturas de dados complexas, pois, ao aplicar este operador a uma variável, o compilador retorna o endereço de memória associado àquela variável, possibilitando seu armazenamento em ponteiros ou seu uso para alocação dinâmica de memória.

MINHAS ANOTAÇÕES



PILHAS E FILAS

MINHAS METAS

- Ter contato com estruturas de dados estáticas.
- Aprender sobre pilhas.
- Aprender a implementar pilhas.
- Aprender sobre filas.
- Aprender a implementar filas.

INICIE SUA JORNADA

Tanto a fila como a pilha são conjuntos ordenados de itens, porém ambas se diferenciam pelas regras de inserção e remoção de elementos. Na pilha, a entrada e a saída de dados se dão pela mesma extremidade, chamada de topo da pilha. Na fila, a entrada e a saída ocorrem em lugares opostos: a entrada acontece no final da fila e a saída no seu início.

Apesar de simples, ambas as estruturas (fila e pilha) são amplamente utilizadas em diversas áreas da computação. Um exemplo pode ser visto no problema de escalonamento do uso do processador descrito por Machado e Maia (2002).

A estrutura, a utilização e as regras de inserção e remoção em pilhas e filas são o foco de estudo deste tema.

PLAY NO CONHECIMENTO

Que tal conhecer a aplicabilidade das pilhas e filas entendendo como elas geralmente são usadas para a construção de sistemas do mundo real? Vamos mergulhar nesse tema nesse podcast! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**



DESENVOLVA SEU POTENCIAL

PILHAS E FILAS

Pilhas

A pilha é uma das estruturas mais simples e mais versáteis dentre as utilizadas na computação. Antes de entrar nas nuances técnicas sobre pilhas, vamos abstrair o seu conceito para uma situação real.

Imagine que você trabalha na construção civil. Existem inúmeros tijolos que precisam ser organizados e preparados para a edificação de um prédio. Você é orientado a empilhá-los próximo ao local da obra. O primeiro tijolo é colocado no chão, no local estipulado, em seguida, o segundo tijolo é colocado em cima do primeiro e cada novo tijolo é colocado no topo da pilha. Na hora de levantar uma nova parede, os tijolos são retirados a partir do topo da pilha de tijolos.

Os tijolos foram empilhados e depois desempilhados. Não faz sentido querer pegar o primeiro tijolo que está lá embaixo, na base, mas sim o primeiro que está livre na parte de cima. Esse é o conceito principal de pilha. É o mesmo para uma pilha de camisas, pilha de caixas de leite, pilha de papéis etc.



Na computação, a pilha é uma estrutura em que os dados são inseridos e removidos no seu topo. São estruturas conhecidas como **Last In, First Out (LIFO)**, que pode ser traduzido por Último a Entrar, Primeiro a Sair.

Vamos, agora, pensar num exemplo para facilitar o entendimento. Temos um vetor de 10 posições no qual serão inseridos os seguintes valores nessa ordem: 1, 5, 12 e 3. O vetor deve ficar com essa cara:

1	5	12	3						
---	---	----	---	--	--	--	--	--	--

Agora serão inseridos mais dois números na sequência: 14 e 2. O vetor ficará com essa configuração:

1	5	12	3	14	2				
---	---	----	---	----	---	--	--	--	--

Pensando que o valor, mais à esquerda, é o começo da pilha, o lado oposto é o seu final e todos os valores vão entrando na primeira casa livre à direita. Esse é o processo de **empilhamento**, em que cada novo valor é inserido **em cima** dos valores previamente inseridos (empilhados).

VOCÊ SABE RESPONDER?

Imagine, agora, que é preciso remover um valor da pilha. Qual será removido e por quê?

O valor removido será o último valor da pilha, posto que, pela regra, o último valor que entra será o primeiro valor a sair (Last In, First Out). É o processo de **desempilhamento**.

1	5	12	3	14					
---	---	----	---	----	--	--	--	--	--

Para construir uma pilha, são necessários pelo menos três elementos: um vetor para armazenar os dados e dois números inteiros, um para marcar o início e outro o final da pilha. Veja o exemplo a seguir em linguagem C da definição de uma estrutura para uma pilha:

```
//Constantes
#define tamanho 10

//Estrutura da Pilha
struct tpilha {
    int dados[tamanho];
    int ini;
    int fim;
};

//Variáveis globais
struct tpilha pilha;
```

Optamos por criar uma constante, chamada *tamanho*, para guardar a capacidade máxima da pilha. Se houver necessidade de aumentar estaticamente o tamanho da pilha, basta alterar o valor da constante sem precisar revisar o resto do código-fonte. Essa constante também será útil na hora de fazer verificações nos processos de empilhamento e desempilhamento.

O vetor *dados* guardará os valores que forem empilhados, o atributo *ini* marca o início da pilha e o atributo *fim*, o seu final. Ambas as variáveis, *ini* e *fim*, são inicializadas com valor zero para indicar que a pilha está vazia.

Em seguida, vamos criar três funções: uma para mostrar o conteúdo da pilha, que ajuda no entendimento e visualização do processo, uma para a entrada (empilhamento) e outra para a saída (desempilhamento).

A função *pilha_mostrar()* é muito simples. Basta um laço de repetição para percorrer todas as posições do vetor e ir imprimindo seus valores na tela. Aqui já usamos a constante *tamanho* para saber quantos valores cabem na pilha.

```
//Mostrar o conteúdo da Pilha
void pilha_mostrar() {
    int i;
    printf("[ ");
    for (i = 0; i < tamanho; i++) {
        printf("%d ", pilha.dados[i]);
    }
    printf("]\n\n");
}
```

Para a entrada dos dados, criamos a função *pilha_entrar()*. Para o empilhamento, é necessário ler o dado diretamente na primeira posição disponível, que representa o topo da pilha. Isso é possível utilizando o atributo *fim* criado na estrutura da pilha. Depois da leitura, o valor de *fim* é atualizado para que ele aponte sempre para a primeira posição disponível.

```
//Adicionar um elemento no final da Pilha
void pilha_entrar(){
    printf("\nDigite o valor a ser empilhado: ");
    scanf("%d", &pilha.dados[pilha.fim]);
    pilha.fim++;
}
```

Do jeito que está, não há nenhum tipo de controle. Os valores são empilhados infinitamente, porém sabemos que a nossa pilha tem um tamanho finito. É necessário criar mecanismos que evitem o estouro da pilha. Vamos escrever novamente a função *pilha_entrar()* adicionando agora um desvio condicional para verificar se existe espaço disponível para o novo empilhamento.

```
//Adicionar um elemento no final da Pilha
void pilha_entrar(){

    if (pilha.fim == tamanho) {

        printf("\nA pilha está cheia, impossível empilhar um novo
               elemento!\n\n");
        system("pause");
    }

    else {

        printf("\nDigite o valor a ser empilhado: ");
        scanf("%d", &pilha.dados[pilha.fim]);
        pilha.fim++;
    }
}
```

Agora faremos uma pequena simulação do funcionamento da função de empilhamento de uma pilha com 5 posições. O vetor dados e as variáveis de controle *ini* e *fim* são inicializados com 0.

Índice	0	1	2	3	4
Dados	0	0	0	0	0

```
ini = 0
fim = 0
tamanho = 5
```

Vamos inserir o número 42 no final da pilha. A primeira coisa que a função *pilha_entrar()* faz é verificar se a pilha atingiu o seu limite. Isso se dá comparando o atributo *fim* com a constante *tamanho*. Como *fim* (0) é diferente de *tamanho* (5), o algoritmo passa para a leitura do dado que será guardado na posição *fim* do vetor *dados*. Em seguida, o atributo *fim* sofre o incremento de 1.

Índice	0	1	2	3	4
Dados	42	0	0	0	0

```
ini = 0
fim = 1
tamanho = 5
```

Vamos inserir mais três valores: 33, 22 e 13. Para cada entrada, será feita a verificação do atributo *fim* com a constante *tamanho*; o valor será inserido no vetor *dados* na posição *fim* e o valor de *fim* será incrementado em 1.

Índice	0	1	2	3	4
Dados	42	33	22	13	0

```
ini = 0
fim = 4
tamanho = 5
```

Note que o atributo *fim* possui valor 4, mas não há nenhum valor inserido nessa posição do vetor *dados*. O atributo *fim* está apontando sempre para a primeira posição disponível no topo da pilha. Ele também representa a quantidade de valores inseridos (4) e que atualmente ocupam as posições 0 a 3 do vetor. Vamos inserir um último número: 9.

Índice	0	1	2	3	4
Dados	42	33	22	13	9

```
ini = 0
fim = 5
tamanho = 5
```

Agora a pilha está completa. Se tentarmos inserir qualquer outro valor, a comparação de *fim* (5) com *tamanho* (5) fará com que seja impresso na tela uma mensagem, informando que a pilha já se encontra cheia, evitando, assim, o seu estouro. O atributo *fim* contém o valor 5, indicando que existem 5 valores no vetor e ele aponta para uma posição inválida, já que um vetor de 5 posições tem índice que começa em 0 e vai até 4.

Para o desempilhamento, vamos criar a função *pilha_sair()*. A remoção se dá no elemento *fim - 1* do vetor *dados*, lembrando que o atributo *fim* aponta para a primeira posição livre, e não é esse o valor que queremos remover, mas sim o valor diretamente anterior. Após a remoção do item, o valor de *fim* deve ser atualizado para apontar corretamente para o final da pilha que acabou de diminuir.

```
//Retirar o último elemento da Pilha  
void pilha_sair() {  
    pilha.dados[pilha.fim-1] = 0;  
    pilha.fim--;  
}
```

O que acontece quando o vetor *dados* está vazio? E se removermos mais elementos do que existem na pilha? Nesse caso, não haverá um estouro na pilha, mas você pode considerar que seria um tipo de implosão. Os vetores não trabalham com índice negativo, e se muitos elementos fossem removidos além da capacidade do vetor, o valor de *fim* iria diminuindo até passar a ser um valor negativo. Na hora da inclusão de um novo valor, ele seria adicionado numa posição inválida e seria perdido.

É preciso fazer um controle antes da remoção para verificar se a pilha está vazia. Vamos comparar, então, o valor de *ini* com *fim*. Se forem iguais, significa que a pilha está vazia e que nenhum valor pode ser removido.

```
//Retirar o último elemento da Pilha
void pilha_sair() {
    if (pilha.ini == pilha.fim) {

        printf("\nA pilha está vazia, não há nada
para desempilhar!\n\n");

        system("pause");
    }
    else {
        pilha.dados[pilha.fim-1] = 0;
        pilha.fim--;
    }
}
```

Vamos resgatar agora a estrutura que carinhosamente empilhamos nas páginas anteriores.

Índice	0	1	2	3	4
Dados	42	33	22	13	9

```
ini = 0
fim = 5
tamanho = 5
```

Precisamos remover um número. Como não se trata de um vetor qualquer, mas sim de uma estrutura em pilha, a remoção começa sempre do último para o primeiro (Last In, First Out). O primeiro passo da função *pilha_sair()* é a comparação dos atributos *ini* (0) e *fim* (5), para verificar se a pilha não está vazia. Como os valores são diferentes, o algoritmo segue para a linha de remoção.

Lembrando que o atributo *fim* aponta para a primeira posição livre (ou seja, o fim da pilha), então, temos que remover o valor do vetor *dados* na posição *fim* subtraindo-se 1, que é a última posição preenchida (topo da pilha). Por último, atualizamos o valor de *fim* para que aponte para a posição recém-liberada do vetor *dados*.

Índice	0	1	2	3	4
Dados	42	33	22	13	0

```
ini = 0
fim = 4
tamanho = 5
```

Vamos, agora, remover os outros valores: 13, 22, 33 e 42. A pilha ficará deste jeito:

Índice	0	1	2	3	4
Dados	0	0	0	0	0

```
ini = 0
fim = 0
tamanho = 5
```

Se tentarmos agora remover mais algum item da pilha, o programa escreverá uma mensagem de erro na tela, já que o atributo *fim* (0) está com o mesmo valor de *ini* (0), que significa que a pilha está vazia.

 ZOOM NO CONHECIMENTO

É muito comum encontrar na literatura os termos **push** para o processo de empilhar e **pop** para o de desempilhar.

A seguir, você encontrará o código-fonte completo de uma pilha em linguagem C. Logo após, encontrará comentários sobre cada bloco do código. Digite o exemplo no seu compilador e execute o programa.

```
1 //Bibliotecas
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <locale.h>
5
6 //Constantes
7 #define tamanho 5
8
9 //Estrutura da Pilha
10 struct tpilha {
11     int dados[tamanho];
12     int ini;
13     int fim;
14 };
15
16 //Variáveis globais
17 struct tpilha pilha;
18 int op;
19
20 //Protipação
21 void pilha_entrar();
22 void pilha_sair();
23 void pilha_mostrar();
24 void menu_mostrar();
25
26 //Função principal
27 int main(){
28     setlocale(LC_ALL, "Portuguese");
29     op = 1;
30     pilha.ini = 0;
31     pilha.fim = 0;
32     while (op != 0) {
33         system("cls");
34         pilha_mostrar();
35         menu_mostrar();
36         scanf("%d", &op);
37         switch (op) {
38             case 1:
39                 pilha_entrar();
40                 break;
41             case 2:
42                 pilha_sair();
43                 break;
44         }
45     }
46     return(0);
}
```

```

47 }
48
49 //Adicionar um elemento no final da Pilha
50 void pilha_entrar() {
51     if (pilha.fim == tamanho) {
52         printf("\nA pilha está cheia, impossível empilhar!\n\n");
53         system("pause");
54     }
55     else {
56         printf("\nDigite o valor a ser empilhado: ");
57         scanf("%d", &pilha.dados[pilha.fim]);
58         pilha.fim++;
59     }
60 }
61
62 //Retirar o último elemento da Pilha
63 void pilha_sair() {
64     if (pilha.ini == pilha.fim) {
65         printf("\nA pilha está vazia, impossível desempilhar!\n\n");
66         system("pause");
67     }
68     else {
69         pilha.dados[pilha.fim-1] = 0;
70         pilha.fim--;
71     }
72 }
73
74 //Mostrar o conteúdo da Pilha
75 void pilha_mostrar() {
76     int i;
77     printf("[ ");
78     for (i = 0; i < tamanho; i++) {
79         printf("%d ", pilha.dados[i]);
80     }
81     printf("]\n\n");
82 }
83
84 //Mostrar o menu de opções
85 void menu_mostrar() {
86     printf("\nEscolha uma opção:\n");
87     printf("1 - Empilhar\n");
88     printf("2 - Desempilhar\n");
89     printf("0 - Sair\n\n");
90 }

```

LINHAS 1A 4

Inclusão de bibliotecas necessárias para o funcionamento do programa.

LINHAS 6 E 7

Definição de constante para o tamanho da pilha.

LINHAS 9 A 14

Registro de estrutura para criar o “tipo pilha” contando com um vetor para armazenar os dados e dois números inteiros para controlar o início e fim da pilha.

LINHAS 16 A 18

Definições de variáveis.

LINHAS 20 A 24

Prototipação das funções.

LINHAS 26 A 47

Função principal (main), a primeira que será invocada na execução do programa.

LINHA 28

Chamada de função para configurar o idioma para português, permitindo o uso de acentos (não funciona em todas as versões do Windows).

LINHAS 29 A 31

Inicialização das variáveis.

LINHAS 32 A 45

Laço principal, que será executado repetidamente até que o usuário decida finalizar o programa.

LINHA 33

Chamada de comando no sistema operacional para limpar a tela.

LINHA 34

Chamada da função que mostra o conteúdo da pilha na tela.

LINHA 35

Chamada da função que desenha o menu de opções.

LINHA 36

Lê a opção escolhida pelo usuário.

LINHAS 37 A 44

Desvio condicional, que faz chamada de acordo com a opção escolhida pelo usuário.

LINHAS 49 A 60

Função *pilha_entrar()*, que faz checagem do topo da pilha e insere novos valores no vetor dados.

LINHAS 62 A 72

Função *pilha_sair()*, que verifica se existem elementos na pilha e remove o último inserido.

LINHAS 74 A 82

Função *pilha_mostrar()*, que lê o conteúdo e desenha o vetor dados na tela.

LINHAS 84 A 90

Função *menu_mostrar()*, que desenha na tela as opções permitidas para o usuário.



FILAS

As filas também são estruturas muito utilizadas, porém suas particularidades fazem com que seu funcionamento seja um pouco menos simples do que o das pilhas.

Voltemos ao exercício de abstração e pensemos como as filas estão presentes no nosso dia a dia. Isso é fácil, o brasileiro adora uma fila e, às vezes, se encontra em uma sem saber para quê. Fila no supermercado, fila no cinema, fila no banco e assim por diante.

Chegando a uma agência bancária, para ser atendido pelo caixa, um cidadão honesto se dirige para o final da fila. Quando um caixa fica livre, aquele que está na fila há mais tempo (primeiro da fila) é atendido.

Esse é conceito básico de toda fila **FIFO (First In, First Out)**, ou na tradução, o Primeiro que Entra é o Primeiro que Sai.

Vamos agora simular uma fila em uma casa lotérica, imaginando que existe lugar apenas para cinco pessoas e que o restante dos apostadores espera fora do edifício. O primeiro a chegar é João, mas ele ainda não é atendido, porque os caixas estão contando o dinheiro e se preparando para iniciar os trabalhos.

João				
------	--	--	--	--

Em seguida, dois clientes entram praticamente juntos, Maria e José, e como todo cavalheiro, José deixa que Maria fique à frente na fila.

João	Maria	José		
------	-------	------	--	--

Um dos funcionários está pronto para iniciar o atendimento e chama o cliente. Qual deles será atendido? O João, porque ele ocupa o primeiro lugar na fila, já que em teoria o primeiro que entra é o primeiro que sai.

	Maria	José		
--	-------	------	--	--

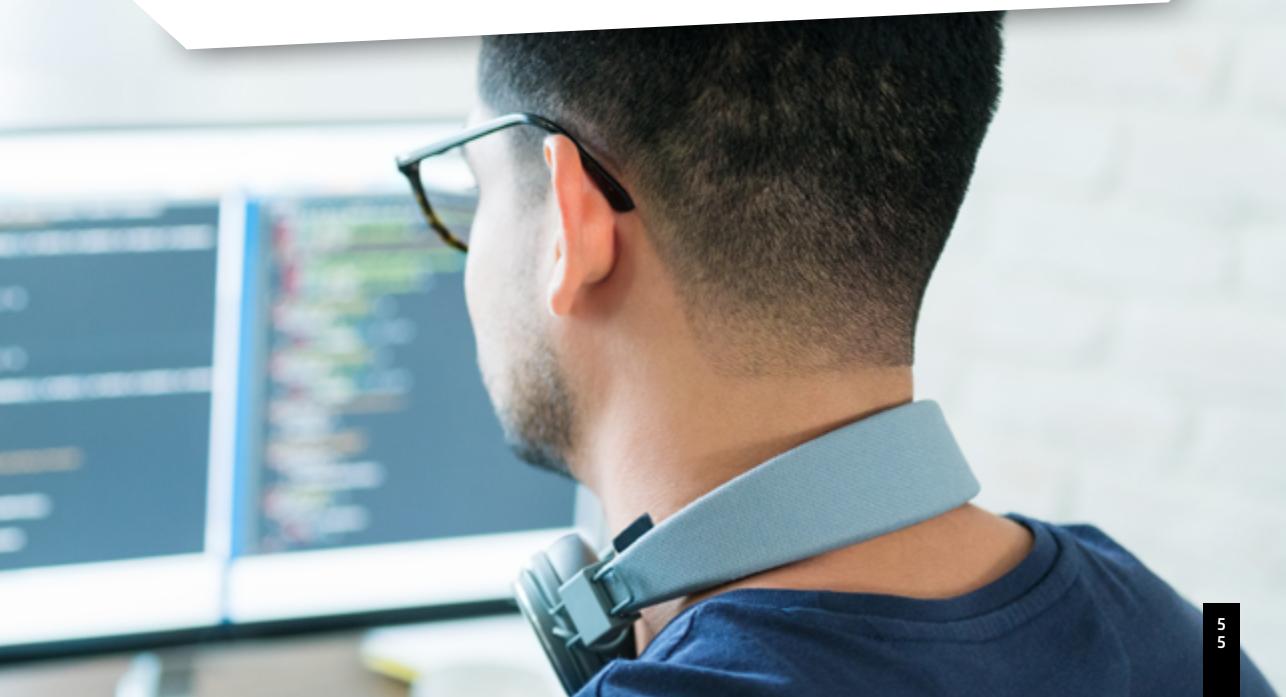
Maria passa automaticamente a ser a primeira da fila, ocupando o lugar que era de João, tendo logo atrás de si José aguardando a sua vez de ser atendido.

Maria	José			
-------	------	--	--	--

Agora que está claro o funcionamento de uma fila, vamos programá-la em linguagem C. A primeira coisa que precisamos é definir a sua estrutura. Usaremos um vetor para armazenar os valores que serão enfileirados e dois números inteiros para fazer o controle de início e fim da fila. Usaremos, também, uma constante para definir a capacidade de armazenamento.

```
//Constantes  
#define tamanho 5  
  
//Estrutura da Fila  
struct tfila {  
    int dados[tamanho];  
    int ini;  
    int fim;  
};  
  
//Variáveis globais  
struct tfila fila;
```

Vamos criar uma função chamada *fila_entrar()* para controlar a entrada de novos valores na fila. A primeira coisa a se fazer é verificar se há espaço. Isso pode ser feito comparando o atributo *fim* com a constante *tamanho*. Caso haja uma posição livre, o valor será inserido no vetor *dados* na posição *fim* e finalmente o valor de *fim* é incrementado em um.



```
//Adicionar um elemento no final da Fila
void fila_entrar() {
    if (fila.fim == tamanho) {
        printf("\nA fila está cheia, impossível adicionar um novo
               valor!\n\n");
        system("pause");
    }
    else {
        printf("\nDigite o valor a ser inserido: ");
        scanf("%d", &fila.dados[fila.fim]);
        fila.fim++;
    }
}
```

Até agora, a fila e a pilha estão muito parecidas na sua definição e modo de entrada de dados. A principal diferença entre as duas estruturas está na forma de saída. Na pilha, sai sempre o elemento mais recente, na fila sai sempre o mais antigo. Assim como na pilha, é necessário fazer uma verificação na fila para saber se ainda existe algum elemento a ser removido.

```
//Retirar o primeiro elemento da Fila
void fila_sair() {
    if (fila.ini == fila.fim) {
        printf("\nA fila está vazia, não há nada para remo-
               ver!\n\n");
        system("pause");
    }
}
```

Como o primeiro elemento da fila será removido, os demais precisam andar em direção ao início, assim como acontece numa fila de verdade. Em seguida, atualizamos o valor do atributo *fim* para apontar corretamente para o final da fila.

```
int i;
for (i = 0; i < tamanho; i++) {
    fila.dados[i] = fila.dados[i+1];
}
fila.dados[fila.fim] = 0;
fila.fim--;
```

A função *fila_sair()* completa fica com essa cara:

```
//Retirar o primeiro elemento da Fila
void fila_sair() {
    if (fila.ini == fila.fim) {
        printf("\nA fila está vazia, não há nada para
remover!\n\n");
        system("pause");
    }
    else {
        int i;
        for (i = 0; i < tamanho; i++) {
            fila.dados[i] = fila.dados[i+1];
        }
        fila.dados[fila.fim] = 0;
        fila.fim--;
    }
}
```

Para finalizar, falta apenas a função *fila_mostrar()*, que possui um laço de repetição que percorre todo o vetor dados e imprime os valores na tela.

```
//Mostrar o conteúdo da Fila
void fila_mostrar() {
    int i;
    printf("[ ");
    for (i = 0; i < tamanho; i++) {
        printf("%d ", fila.dados[i]);
    }
    printf("]\n\n");
}
```

A principal regra para uma fila é que o primeiro que entra é o primeiro que sai. Esse exemplo não é a única forma de implementação de fila. No nosso caso, cada vez que alguém sai da fila, todos os outros clientes precisam se mover para a primeira posição que ficou livre à sua esquerda.

Existem outras formas de fila, como a fila cílica. Ao invés de mover os dados para a esquerda sempre que uma posição fica livre, move-se o atributo que marca o início da fila para a direita. Essa é uma alternativa interessante para quando se tem filas muito grandes e não se pode perder tempo movendo todo o resto dos dados em direção ao começo da fila.

Vamos voltar ao exemplo da fila anterior, com o João, a Maria e o José.

João	Maria	José		
------	-------	------	--	--

```
ini = 0
fim = 3
tamanho = 5
```

Quando o João deixa a fila, não é a Maria que anda em direção à posição ocupada por João, mas é o início da fila que se move em direção à Maria.

	Maria	José		
--	-------	------	--	--

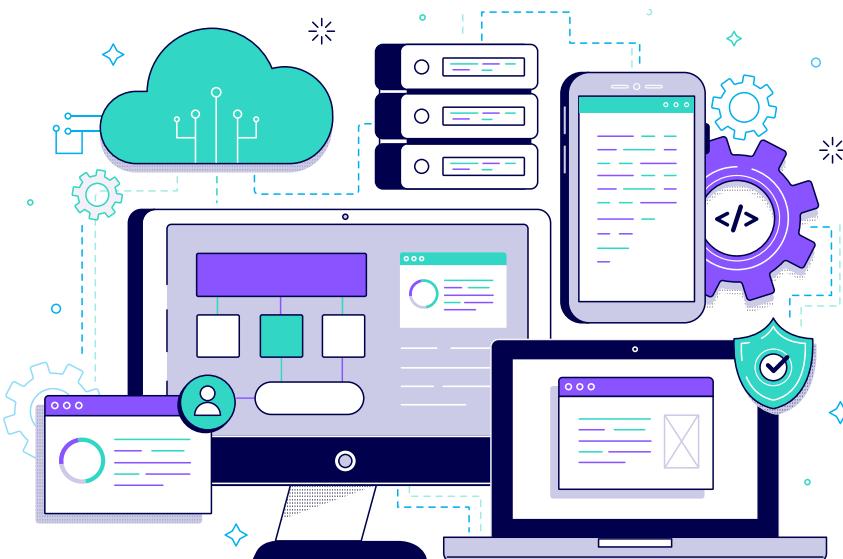
```
ini = 1
fim = 3
tamanho = 5
```

Nesse exemplo, para saber o tamanho da fila, é necessário subtrair o valor do atributo *fim* (3) do atributo *ini* (1).

Apesar de ser implementado de forma diferente, o conceito ainda é o mesmo: **o primeiro que entra é o primeiro que sai**. Ninguém gosta de fura-fila, não é verdade?

ZOOM NO CONHECIMENTO

É comum, na literatura, o uso dos termos enqueue (ou push_back), para indicar a inserção, e dequeue (ou pop), para a remoção em filas.



A seguir, temos o código-fonte completo de uma fila em linguagem C. Logo após, há comentários explicando cada bloco do código. Digite o exemplo no seu compilador e execute o programa.

```
1 //Bibliotecas
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <locale.h>
5
6 //Constantes
7 #define tamanho 5
8
9 //Estrutura da Fila
10 struct tfila {
11     int dados[tamanho];
12     int ini;
13     int fim;
14 };
15
16 //Variáveis globais
17 struct tfila fila;
18 int op;
19
20 //Prototipação
21 void fila_entrar();
22 void fila_sair();
23 void fila_mostrar();
24 void menu_mostrar();
25
26 //Função principal
27 int main(){
28     setlocale(LC_ALL, "Portuguese");
29     op = 1;
30     fila.ini = 0;
31     fila.fim = 0;
32     while (op != 0) {
33         system("cls");
34         fila_mostrar();
35         menu_mostrar();
36         scanf("%d", &op);
37         switch (op) {
38             case 1:
39                 fila_entrar();
40                 break;
41             case 2:
42                 fila_sair();
43                 break;
44         }
45     }
46     return(0);
47 }
48
49 //Adicionar um elemento no final da Fila
50 void fila_entrar(){
```

```

50 void fila_entrar() {
51     if (fila.fim == tamanho) {
52         printf("\nA fila está cheia, volte outro dia!\n\n");
53         system("pause");
54     }
55     else {
56         printf("\nDigite o valor a ser inserido: ");
57         scanf("%d", &fila.dados[fila.fim]);
58         fila.fim++;
59     }
60 }
61
62 //Retirar o primeiro elemento da Fila
63 void fila_sair() {
64     if (fila.ini == fila.fim) {
65         printf("\nFila vazia, mas logo aparece alguém!\n\n");
66         system("pause");
67     }
68     else {
69         int i;
70         for (i = 0; i < tamanho; i++) {
71             fila.dados[i] = fila.dados[i+1];
72         }
73         fila.dados[fila.fim] = 0;
74         fila.fim--;
75     }
76 }
77
78 //Mostrar o conteúdo da Fila
79 void fila_mostrar() {
80     int i;
81     printf("[ ");
82     for (i = 0; i < tamanho; i++) {
83         printf("%d ", fila.dados[i]);
84     }
85     printf("]\n\n");
86 }
87
88 //Mostrar o menu de opções
89 void menu_mostrar() {
90     printf("\nEscolha uma opção:\n");
91     printf("1 - Incluir na Fila\n");
92     printf("2 - Excluir da Fila\n");
93     printf("0 - Sair\n\n");
94 }

```

LINHAS 1A 4

Inclusão de bibliotecas necessárias para o funcionamento do programa.

LINHAS 6 E 7

Definição de constante para o tamanho da pilha.

LINHAS 9 A 14

Registro de estrutura para criar o “tipo fila” contando com um vetor para armazenar os dados e dois números inteiros para controlar o início e fim da fila.

LINHAS 16 A 18

Definições de variáveis.

LINHAS 20 A 24

Prototipação das funções.

LINHAS 26 A 47

Função principal (main), a primeira que será invocada na execução do programa.

LINHA 28

Chamada de função para configurar o idioma para português, permitindo o uso de acentos (não funciona em todas as versões do Windows).

LINHAS 29 A 31

Inicialização das variáveis.

LINHAS 32 A 45

Laço principal, que será executado repetidamente até que o usuário decida finalizar o programa.

LINHA 33

Chamada de comando no sistema operacional para limpar a tela.

LINHA 34

Chamada da função que mostra o conteúdo da Fila na tela.

LINHA 35

Chamada da função que desenha o menu de opções.

LINHA 36

Lê a opção escolhida pelo usuário.

LINHAS 37 A 44

Desvio condicional, que faz chamada de acordo com a opção escolhida pelo usuário.

LINHAS 49 A 60

Função *fila_entrar()*, que faz checagem do fim da fila e insere novos valores no vetor dados.

LINHAS 62 A 76

Função *fila_sair()*, que verifica se existem elementos na fila e remove o elemento mais antigo.

LINHAS 78 A 86

Função *fila_mostrar()*, que lê o conteúdo e desenha o vetor dados na tela.

LINHAS 88 A 94

Função *menu_mostrar()*, que desenha na tela as opções permitidas para o usuário.



INDICAÇÃO DE LIVRO

Ciência da Computação.

Sobre o Livro: são estudados vários tipos de estruturas de dados, sua aplicação e manipulação. A escolha da estrutura de dados adequada para representar uma realidade deve considerar aspectos como alocação da memória, formas de consulta, acesso e operações de inserção e exclusão. O objetivo da obra é apresentar as principais estruturas de dados conhecidas de uma forma prática e fácil de compreender. São apresentadas as diversas aplicações dessas estruturas por meio de exemplos de programas em C e em Pascal.



EM FOCO

Assista às aulas para compreender mais sobre o tema. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

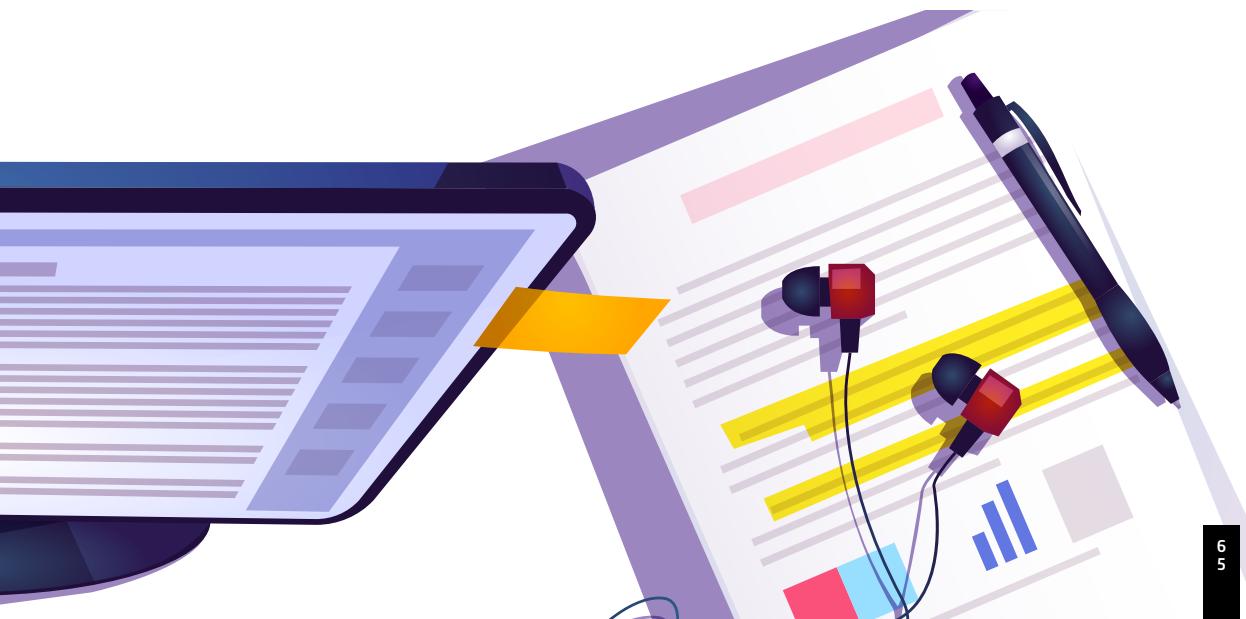


NOVOS DESAFIOS

Tanto as pilhas como as filas são tipos de listas, ou seja, coleção de dados agrupados. A diferença é que tanto a pilha como a fila possuem regras de entrada e saída, diferente das listas simples.

As pilhas são usadas em casos em que é mais importante resolver o problema mais recente, ou mais oneroso, ou mais próximo. As filas têm função contrária e são aplicadas na solução de casos em que é proibido que questões mais recentes recebam atenção antes de questões mais antigas.

As pilhas e filas são estruturas de dados fundamentais na computação e são amplamente utilizadas no mercado de trabalho para resolver uma variedade de problemas, como a navegação em aplicativos e websites onde as pilhas são usadas para gerenciar o histórico de páginas visitadas, permitindo que o usuário retorne à página anterior com facilidade. Já as filas podem ser usadas em sistemas de atendimento ao cliente onde as solicitações dos clientes são colocadas em uma fila e tratadas na ordem em que foram recebidas.



VAMOS PRATICAR

1. Quando um livro é devolvido na biblioteca, o funcionário responsável pelo recebimento coloca o livro em cima de uma pilha de livros na mesa ao lado da recepção. O auxiliar de bibliotecário pega o livro do topo da pilha, verifica o seu código e leva-o para o seu devido local no acervo.

No atual sistema de informação, é possível verificar se o livro está disponível ou se está emprestado. Entretanto, o livro que acabou de ser devolvido ainda não se encontra na prateleira, pois existe um intervalo de tempo entre a devolução dele e o momento em que ele é guardado na estante.

A sugestão do departamento de TI é de criar um programa que faça o controle na pilha, assim, pode-se verificar se o livro ainda não foi guardado e qual a sua posição dentro da pilha de livros que aguardam ao lado da recepção.

- a) Crie uma estrutura para a pilha de livros. Lembre-se de que ela tem que ter um vetor para armazenar os dados (código, nome do livro e autor) e dois números inteiros, um para controlar o início e outro o final da pilha.
b) Defina a variável que será um vetor do tipo pilha de livros.
c) Faça uma função de empilhamento, lembrando-se de verificar se a pilha atingiu o tamanho máximo de livros (a mesa não aguenta muito peso).
d) Crie uma função para desempilhamento de livros. Não se esqueça de que é necessário verificar se ainda existem livros para serem guardados.
e) Elabore uma função que apresente na tela a lista de todos os livros que se encontram empilhados ao lado da recepção.
2. Uma agência bancária quer inovar seu atendimento, criando mais conforto para seus clientes. Para isso, foram colocadas diversas cadeiras na recepção do banco. Quando um cliente chega, o atendente lança no computador o seu nome e o horário que chegou. Assim que um caixa fica livre, a recepcionista olha no sistema e chama o primeiro cliente da fila. Dessa forma, é possível que os clientes esperem confortavelmente sentados pelo seu atendimento, não importando o local onde se encontram dentro da agência bancária.
a) Faça uma estrutura para o controle da fila. Você precisa guardar o nome e a hora que o cliente chegou. Use um vetor para armazenar os dados e dois números inteiros, um para controlar o início e outro o final da fila.
b) Defina a variável que será um vetor do tipo fila de clientes.
c) Crie uma função enfileirar, lembrando que é preciso verificar se há espaço na fila (o número de cadeiras na recepção é limitado).

VAMOS PRATICAR

- d) Elabore a função desenfileirar cliente, não se esqueça de que é necessário verificar se ainda existem clientes para serem atendidos.
 - e) Faça uma função que apresente na tela a lista de todos os clientes que estão aguardando atendimento na recepção.
3. Em muitos aplicativos, como editores de texto, softwares de design gráfico ou ambientes de desenvolvimento, as ações do usuário são armazenadas em uma pilha para permitir operações de desfazer e refazer em ordem reversa (CORMEN, 2002).

Fonte: CORMEN, T. H. et al. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. v. 2.

Com isso em mente, indique qual é o princípio fundamental das pilhas?

- a) Princípio do “último a entrar, primeiro a sair” (LIFO – Last In, First Out).
 - b) Princípio do “primeiro a entrar, primeiro a sair” (FIFO – First In, First Out).
 - c) Princípio da “ordem alfabética inversa” (ZLFO – Zast In, Last Out).
 - d) Princípio da “ordem inversa de chegada” (RIFO – Reverse In, First Out).
 - e) Princípio do “último a entrar, último a sair” (LILO – Last In, Last Out).
4. Suponha que você está desenvolvendo um sistema de atendimento de pedidos de um restaurante. Nesse sistema, os pedidos são colocados em uma fila e atendidos na ordem de chegada. Cada pedido é representado por um número inteiro positivo, e os pedidos são processados de acordo com o princípio “primeiro a entrar, primeiro a sair” (FIFO – First In, First Out).

No início do dia, a fila está vazia. Ao longo do dia, os seguintes pedidos são registrados no sistema, seguindo a ordem de chegada:

Pedido A
Pedido B
Pedido C
Pedido D
Pedido E

Com base nas informações fornecidas, indique a resposta às duas perguntas abaixo, respectivamente:

- I - Qual é o primeiro pedido a ser atendido?
- II - Qual é o último pedido a ser atendido?

VAMOS PRATICAR

- a) I- Pedido A; II- Pedido E.
 - b) I- Pedido E; II- Pedido A.
 - c) I- Pedido A; II- Pedido B.
 - d) I- Pedido E; II- Pedido D.
 - e) I- Pedido B; II- Pedido D.
5. As estruturas de dados são fundamentais na Ciência da Computação, permitindo a organização eficiente e o acesso aos dados armazenados. Duas estruturas amplamente utilizadas são as pilhas e as filas, que desempenham papéis cruciais em diversas aplicações, desde sistemas operacionais até algoritmos de processamento de dados em larga escala. Ao dominar essas estruturas, os programadores podem criar soluções mais elegantes e eficazes para uma variedade de problemas computacionais (CORMEN, 2002).

Fonte: CORMEN, T. H. *et al.* **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. v. 2.

A seguir, são apresentadas quatro asserções sobre pilhas e filas. Analise cada asserção e marque a opção correta:

- I - As pilhas e filas são estruturas de dados lineares.
- II - O princípio de acesso às filas é o “primeiro a entrar, primeiro a sair” (FIFO - First In, First Out).
- III - O princípio de acesso às pilhas é o “último a entrar, primeiro a sair” (LIFO - Last In, First Out).
- IV - Tanto pilhas quanto filas permitem adicionar elementos apenas no começo da estrutura.

É correto o que se afirma em:

- a) I e IV, apenas.
- b) II e III, apenas.
- c) III e IV, apenas.
- d) I, II e III, apenas.
- e) II, III e IV, apenas.

REFERÊNCIAS

MACHADO, F. B.; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 3. ed. Rio de Janeiro: LTC, 2002.

GABARITO

1. a)

```
#include <string.h>

//Constantes
#define tamanho 5

//Estrutura do Livro
struct tlivro {
    int codigo;
    char nome[50];
    char autor[50];
};

//Estrutura da Pilha
struct tpilha {
    tlivro dados[tamanho];
    int ini;
    int fim;
};
```

GABARITO

b)

```
//Variáveis globais  
tpilha pilha;
```

c)

```
//Adicionar um elemento no final da Pilha  
void pilha_entrar(){  
    if (pilha.fim == tamanho) {  
        printf("\nA pilha está cheia, impossível empilhar!\n\n");  
        system("pause");  
    }  
    else {  
        printf("\nDigite o código do livro: ");  
        scanf("%d", &pilha.dados[pilha.fim].codigo);  
        printf("\nDigite o nome do livro: ");  
        scanf("%s", pilha.dados[pilha.fim].nome);  
        printf("\nDigite o nome do autor: ");  
        scanf("%s", pilha.dados[pilha.fim].autor);  
        pilha.fim++;  
    }  
}
```

GABARITO

d)

```
//Retirar o último elemento da Pilha

void pilha_sair() {
    if (pilha.ini == pilha.fim) {
        printf("\nA pilha está vazia, impossível desem-
               pilhar!\n\n");
        system("pause");
    }
    else {
        pilha.dados[pilha.fim-1].codigo = 0;
        strcpy(pilha.dados[pilha.fim-1].nome, "");
        strcpy(pilha.dados[pilha.fim-1].autor, "");
        pilha.fim--;
    }
}
```

e)

```
//Mostrar o conteúdo da Pilha

void pilha_mostrar() {
    int i;
    printf("[ ");
    for (i = 0; i < tamanho; i++) {
        printf("%d ", pilha.dados[i].codigo);
    }
    printf("]\n\n");
}
```

GABARITO

2. a)

```
#include <string.h>

//Constantes
#define tamanho 5

//Estrutura do Cliente
struct tcliente {
    char nome[50];
    char hora[6];
};

//Estrutura da Fila
struct tfila {
    struct tcliente dados[tamanho];
    int ini;
    int fim;
};
```

b)

```
//Variáveis globais
struct tfila fila;
```

GABARITO

c)

```
//Adicionar um elemento no final da Fila
void fila_entrar(){
    if (fila.fim == tamanho) {
        printf("\nA fila está cheia, volte outro dia!\n\n");
        system("pause");
    }
    else {
        printf("\nDigite o nome do cliente que chegou:");
        scanf("%s", fila.dados[fila.fim].nome);
        printf("\nDigite a hora da chegada do cliente:");
        scanf("%s", fila.dados[fila.fim].hora);
        fila.fim++;
    }
}
```

GABARITO

d)

```
//Retirar o primeiro elemento da Fila
void fila_sair() {
    if (fila.ini == fila.fim) {
        printf("\nFila vazia, mas logo aparece alguém!\n\n");
        system("pause");
    }
    else {
        int i;
        for (i = 0; i < tamanho; i++) {
            strcpy(fila.dados[i].nome, fila.dados[i+1].nome);
            strcpy(fila.dados[i].hora, fila.dados[i+1].hora);
        }
        strcpy(fila.dados[fila.fim].nome, "");
        strcpy(fila.dados[fila.fim].hora, "");
        fila.fim--;
    }
}
```

GABARITO

e)

```
//Mostrar o conteúdo da Fila  
void fila_mostrar() {  
    int i;  
    printf("[ ");  
    for (i = 0; i < tamanho; i++) {  
        printf("Cliente %s ", fila.dados[i].nome);  
        printf("chegou as %s horas \n", fila.dados[i].hora);  
    }  
    printf("]\n\n");  
}
```

3. Opção A. Uma pilha é uma estrutura de dados linear que segue o princípio fundamental do “último a entrar, primeiro a sair” (LIFO – Last In, First Out).

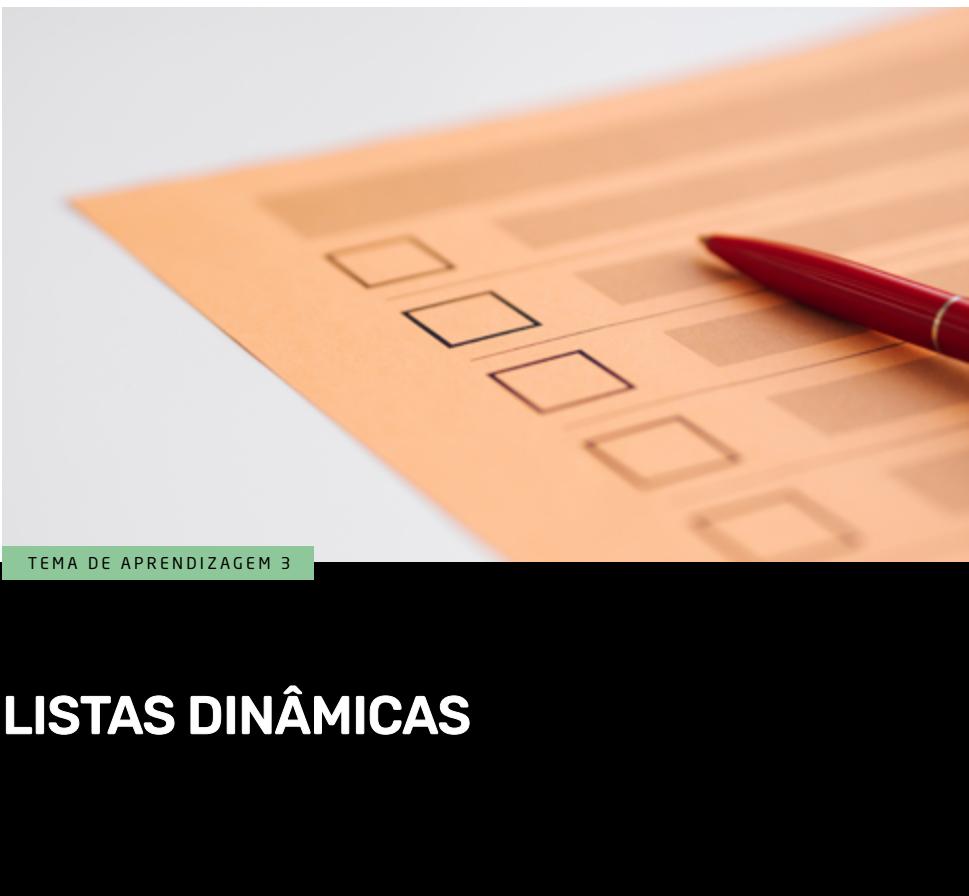
Essa característica define que o último elemento inserido na pilha é o primeiro a ser removido. Como resultado, os elementos são empilhados e desempilhados na ordem inversa à qual foram inseridos. Quando um novo elemento é adicionado à pilha, ele é colocado no topo da estrutura, tornando-se o elemento de acesso mais fácil. Qualquer operação de remoção ou acesso a elementos na pilha sempre será realizada no topo. Isso cria um comportamento semelhante ao de uma pilha de pratos, onde você adiciona um prato no topo e remove o prato mais recentemente adicionado primeiro. Sendo assim, os demais princípios não se aplicam à estrutura de pilha.

4. Opção A. A única alternativa que atende o princípio “primeiro a entrar, primeiro a sair” (FIFO – First In, First Out) é alternativa A: “1) Pedido A; 2) Pedido E”, pois o primeiro a entrar na fila foi o pedido A, logo, ele será feito primeiramente. Como o último pedido a chegar foi o pedido E, ele será o último a ser feito.

GABARITO

5. Opção D.

- I) A asserção I é verdadeira. Tanto pilhas quanto filas são exemplos de estruturas de dados lineares, onde os elementos são organizados em uma sequência linear.
- II) A asserção II é verdadeira. O princípio de acesso às filas é o “primeiro a entrar, primeiro a sair” (FIFO - First In, First Out), onde o primeiro elemento adicionado é o primeiro a ser removido.
- III) A asserção III é falsa. O princípio de acesso às pilhas é o “primeiro a entrar, primeiro a sair” (FIFO - First In, First Out), assim como nas pilhas, e não o “último a entrar, primeiro a sair” (LIFO - Last In, First Out) como afirmado na asserção.
- IV) A asserção IV é falsa. Tanto pilhas quanto filas permitem adicionar elementos apenas no final da estrutura.



LISTAS DINÂMICAS

MINHAS METAS

- Aprender o conceito de listas.
- Entender o que são listas simples.
- Entender o que são listas encadeadas, duplamente encadeadas e circulares.
- Usar a alocação dinâmica de memória na construção de listas.
- Entender como se dá a navegação em listas.

INICIE SUA JORNADA

Apesar da facilidade e praticidade dos vetores, esse tipo de dado possui diversas limitações. Por este motivo, neste tema veremos uma nova estrutura chamada **Lista**. Apesar de ser parecida com pilhas e filas, o seu conceito não inclui regras de entrada e saída como as que existem em estruturas como a FIFO e a LIFO. Todavia, as listas possuem características próprias que lhe dão grande **versatilidade**.

Apesar de ser possível criar uma lista de forma estática, focaremos, neste tema, na criação e na aplicação das listas dinâmicas.

Entre as estruturas de dados existentes, as listas dinâmicas aparecem como um elemento intrigante. Podemos nos perguntar: como lidar com a necessidade de armazenar uma coleção de elementos que pode crescer ou diminuir ao longo do tempo? Listas dinâmicas surgem como uma solução, uma vez que alocam memória conforme necessário e evitam o desperdício de recursos.

As listas dinâmicas representam uma abstração poderosa, em que cada nó não é apenas um ponto no espaço, mas sim um elo entre elementos. Cada nó contém o valor do elemento e uma referência ao próximo nó na sequência. Essa relação é o ponto principal das listas dinâmicas: a conexão que liga cada elemento ao próximo.

Entretanto, será que as listas dinâmicas são úteis no nosso dia a dia como programadores? Pense em um editor de texto: nesse tipo de aplicação, o recurso de desfazer é essencial. Será que faz sentido usarmos uma lista dinâmica para estruturar essa funcionalidade? Durante a leitura deste tema, pense nessa possibilidade!



PLAY NO CONHECIMENTO

Qual será a aplicabilidade das listas no mundo real? De simples listas de tarefas à navegação em músicas e loops de ações em jogos, as listas são estruturas de dados poderosas e muito utilizadas. Venha aprender mais sobre nesse podcast!

Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.



DESENVOLVA SEU POTENCIAL

LISTAS DINÂMICAS

Fundamentos de Listas Dinâmicas

A maneira mais simples de guardar um conjunto de informação na memória se dá pelo uso de vetores. Devido às suas características, o vetor mantém os dados armazenados numa estrutura sequencial. Imaginemos uma variável do tipo vetor de inteiros chamada *vec*. Para acessarmos a quinta posição desse vetor, precisamos apenas usar o índice correto na leitura do vetor *vec[40]* (Figura 1), lembrando que, em C, um vetor começa na posição 0 e vai até $n-1$, em que n é a quantidade de elementos no vetor.

Vetor vec						
Índices →	0	1	2	3	4	5
Valores →	15	22	50	13	42	77

Figura 1 - Acesso à quinta posição do vetor *vec* / Fonte: o autor.

Descrição da Imagem: tabela com três linhas e seis colunas. A primeira linha preenche todas as colunas e está escrita Vetor *vec*, a linha dois indica os Valores e a linha três os Índices. Na primeira coluna, o índice é zero e o valor é quinze, na segunda coluna o índice é um e o valor é vinte e dois, na terceira coluna o índice é dois e o valor é cinquenta, na quarta coluna o índice é três e o valor é treze, na quinta coluna o índice é quatro e o valor é quarenta e dois e na sexta e na última coluna o índice é cinco e o valor é setenta e sete. Fim da descrição.

Na criação de uma pilha, o vetor se mostra muito eficaz, pois a entrada e a saída são feitas sempre no seu último elemento. Para a fila, o vetor mostra algumas dificuldades de implementação, posto que o elemento que sai é sempre o mais antigo e uma das formas de manter os dados de forma sequencial é mover todo o resto dos dados para as posições precedentes.

Não é um problema quando se trabalha com poucos dados, mas imagine uma fila com 100, 500, 1.000 ou mais posições. Dá para **mover** o início da fila alterando o valor de uma variável auxiliar, criada para indicar qual a posição de início da fila no vetor, mas aí o início da fila não seria no início do vetor, perdendo, assim, a forma sequencial da estrutura.

É preciso, então, alterar a forma de organização dos dados, para garantir que a ordenação seja independente do índice da variável. Suponhamos que os itens de uma pilha ou fila contivessem, além do seu valor, o endereço do próximo elemento. Dessa forma, a partir do primeiro elemento da estrutura, seria possível percorrê-la toda na ordem correta, mesmo que, fisicamente, os dados não se encontrem ordenados na memória do computador. Segundo Tenenbaum (1995), tal ordenação explícita faz surgir uma estrutura de dados que é conhecida como lista ligada linear. Na literatura, também podem-se encontrar referências à tal estrutura como lista encadeada.

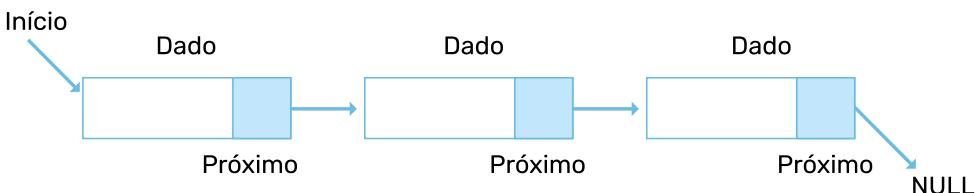


Figura 2 - Representação de lista encadeada / Fonte: a autora.

Descrição da Imagem: a figura contém três retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. Uma seta está apontada para o primeiro nó indicando o início da lista e, no final, outra seta apontando para o nó à direita. O segundo nó aponta para o terceiro. O terceiro e último nó aponta para a palavra NULL indicando o fim da lista. Fim da descrição.

Cada item na lista é chamado de **nó** e contém pelo menos dois elementos: um de dados e um de endereço. O campo de **dados** contém o real elemento da lista e o campo de **endereço** é um ponteiro para o próximo nó. É preciso, também,

uma variável extra que contenha o endereço do primeiro elemento da lista. Para finalizar uma lista, o atributo endereço do último nó contém um valor especial conhecido como null (nulo).

Existem, também, outros tipos de listas. Uma delas, que é muito conhecida, é a lista duplamente encadeada (Figura 3). Cada nó possui, pelo menos, três campos: um de dados e dois de endereço. Um dos endereços é usado para apontar ao nó anterior e o segundo aponta para o próximo nó. Dessa forma, pode-se percorrer a lista em ambas as direções, algo muito versátil e que não é possível na lista simples. O endereço anterior do primeiro nó também contém um valor null indicando o início da lista.

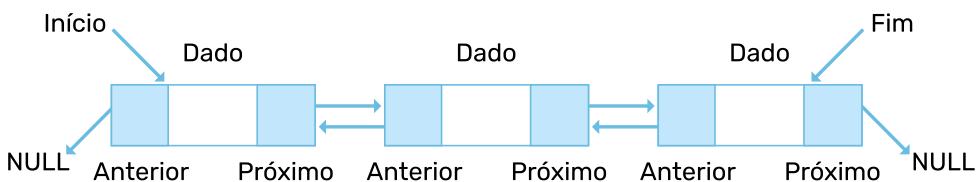


Figura 3 - Representação de lista duplamente encadeada / Fonte: o autor.

Descrição da Imagem: a figura contém três retângulos que representam nós. Cada nó é dividido em três partes, o meio representa o dado a ser armazenado, o início representa o ponteiro para o nó anterior e o fim o ponteiro para o próximo nó. No início do primeiro nó uma seta aponta para ele indicando o início e uma seta sai dele apontando para a palavra NULL, indicando que ele pode ser o início ou o fim da lista. O nó do meio tem uma seta apontando para o nó anterior e uma seta sendo apontada do nó anterior para ele, o mesmo acontece do segundo para o terceiro nó. O terceiro e último nó tem também uma seta apontada para o ponteiro da direita indicando o Fim e uma seta saindo do mesmo ponteiro apontando para a palavra NULL indicando o fim da lista. Fim da descrição.

Outro tipo de listas são as **circulares**, que são estruturas de dados que **não têm fim**. É como se fosse uma lista ligada linear simples, porém, em vez de o último elemento apontar para null, ele aponta para o **primeiro** elemento da lista. Seria possível, ainda, combinar o conceito de lista circular com lista duplamente encadeada. Na Figura 4, do lado esquerdo, temos uma lista encadeada circular simples; do lado direito, uma lista circular duplamente encadeada.

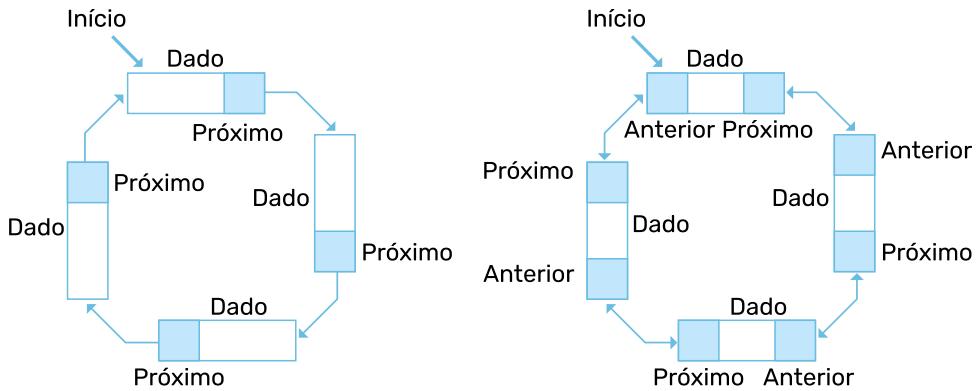


Figura 4 - Representação de uma lista encadeada circular simples e de uma lista circular duplamente encadeada / Fonte: o autor.

Descrição da Imagem: a figura está dividida em duas partes. A primeira contém quatro retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. Uma seta aponta do ponteiro do primeiro nó para o dado do segundo e assim sucessivamente até que o quarto nó aponta para o primeiro, formando um círculo. O primeiro nó também tem uma seta apontando para a região do dado indicando o início. A segunda parte da imagem replica a primeira, mas o nó agora contém três partes, o meio representa o dado a ser armazenado, o início representa o ponteiro para o nó anterior e o fim o ponteiro para o próximo nó. As setas que o relacionam agora apontam para os dois lados, mostrando que a lista pode ser percorrida em ambas as direções. Fim da descrição.

No caso do vetor, todo o espaço é reservado na memória pelo compilador, independentemente se estamos usando uma, nenhuma ou todas as suas posições. Já com uma lista dinâmica, isso não ocorre, pois vamos adicionando nós sempre que for necessário.

A **inserção** de um novo elemento numa lista é simples (Figura 5). Basta encontrar a posição desejada, fazer com que o nó atual aponte para o nó que será inserido e o novo nó aponte para o local onde apontava o nó imediatamente anterior. A imagem a seguir ilustra bem esse conceito.

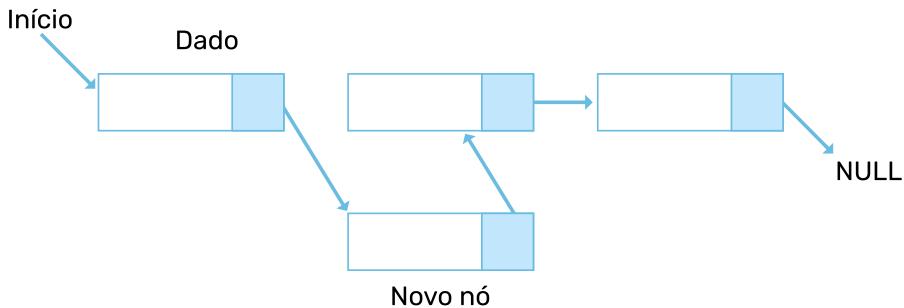


Figura 5 - Representação de uma inserção em uma lista / Fonte: o autor.

Descrição da Imagem: a figura é composta na parte superior por três retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. No início do primeiro nó, uma seta aponta para ele indicando o início e no fim uma seta apontando para um novo nó na parte inferior da imagem, indicando a inserção de um novo nó, que tem uma seta saindo do ponteiro e se conectando ao segundo nó da parte superior. O segundo nó, por sua vez, se conecta com o início do terceiro e tem uma seta no final apontando para a palavra NULL, que indica seu fim. Fim da descrição.

O processo de **remoção** também é simples (Figura 6), basta que o nó anterior ao que for removido passe a apontar para o elemento que o nó removido apontava.

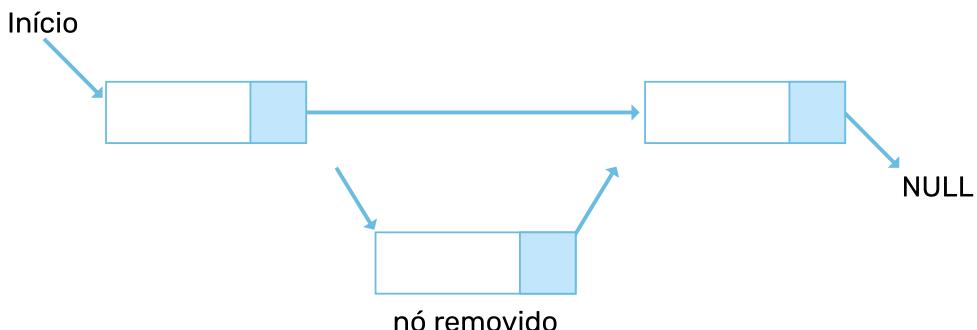


Figura 6 - Representação de uma remoção em uma lista / Fonte: o autor.

Descrição da Imagem: a figura é composta na parte superior por dois retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. Os nós são conectados por uma seta ligando o ponteiro do primeiro e o dado no segundo, no primeiro nó uma seta apontando para ele indica o início e no fim do segundo nó uma seta aponta para a palavra NULL, indicando o fim. Na parte inferior da imagem tem outro nó, com uma seta apontando para o início, sem conectar com o primeiro nó superior e uma seta saindo do fim do nó inferior apontando para o início do segundo nó superior, mas sem encostar, indicando que o nó estava entre os dois superiores, mas foi removido. Fim da descrição.

Implementando uma Lista Dinâmica

A primeira coisa que precisamos para criar uma lista dinâmica é a estrutura do nó. Ela terá dois elementos: um inteiro, que guardará a informação propriamente dita, e um ponteiro, que aponta para o próximo nó.

```
struct no {  
    int dado;  
    struct no *proximo;  
};
```

Com a estrutura já definida, precisamos criar agora um ponteiro do tipo nó, que será necessário para fazer a alocação dinâmica na memória para cada novo elemento da lista.

```
typedef no *ptr_no;
```

Por último, criaremos uma variável que irá apontar para o início da lista. A partir dela, poderemos navegar do primeiro ao último elemento, fazer remoções e inserções.

```
ptr_no lista;
```

A variável do tipo ponteiro lista foi criada para apontar para a nossa lista encadeada. Como ela ainda não existe, vamos criar o primeiro nó com o atributo *dado* valendo 0 e o ponteiro *proximo* apontando para null.

```
lista = (ptr_no) malloc(sizeof(no));  
lista->dado = 0;  
lista->proximo = NULL;
```

A função *malloc* precisa, como parâmetro de *sizeof*, do tipo de dado que será alocado (estrutura *no*); com isso, o compilador saberá a quantidade exata de memória que precisará reservar. O retorno da função precisa ser do tipo esperado pela variável lista, que no caso é do tipo *ptr_no*.

Com a lista definida e devidamente inicializada, podemos criar agora a função *lista_mostrar()*, que será utilizada para desenhar na tela o conteúdo da lista dinâmica. Ela recebe, como parâmetro, uma variável do tipo *ptr_no*, que será aquela que criamos lá no começo para apontar para o início da lista.

```
void lista_mostrar(ptr_no lista) {
    system("cls");
    while(1) {
        printf("%d, ", lista->dado);
        if (lista->proximo == NULL) {
            break;
        }
        lista = lista->proximo;
    }
}
```

Nesse exemplo, criamos um laço infinito (*loop* infinito) que irá funcionar até que o atributo *próximo* do nó atual seja nulo. Essa abordagem é perigosa, porque, se houver um erro na implementação e o valor do último elemento não apontar para um endereço nulo, ficaremos presos nesse laço de forma indeterminada.

Vamos, então, redesenhar a função *lista_mostrar()*, fazendo com que a verificação da nulidade do atributo *proximo* esteja definida como condicional do laço.

```

void lista_mostrar_2(ptr_no lista) {
    system("cls");
    while(lista->proximo != NULL) {
        printf("%d, ", lista->dado);
        lista = lista->proximo;
    }
    printf("%d, ", lista->dado);
}

```

A repetição irá parar quando chegar ao último elemento, mas sairá do laço antes de imprimir o valor de dado do último nó na tela. Para isso, faz-se necessário um comando *printf* adicional no final da função.

Para a inserção dinâmica de um novo nó na lista, vamos criar uma função chamada *lista_inserir()*. Sabemos onde a nossa lista dinâmica começa, pois temos uma variável do tipo ponteiro chamada *lista*, que foi criada especialmente para isso. A partir dela, vamos percorrer toda a estrutura até achar o valor null no elemento *proximo* do último nó (Figura 7).

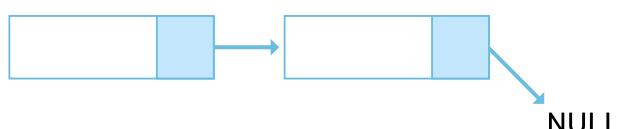


Figura 7 - Representação do último nó da lista / Fonte: o autor.

Descrição da Imagem: a figura é composta na parte superior por dois retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. O ponteiro do primeiro nó se liga por uma seta com o início do segundo e o ponteiro do segundo nó tem uma seta apontada para a palavra NULL, indicando o fim. Fim da descrição.

Criamos um novo nó e fazemos o último nó da lista apontar para ele (Figura 8).



Figura 8 - Representação da criação de um novo nó / Fonte: o autor.

Descrição da Imagem: a figura é composta na parte superior por três retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. O ponteiro do primeiro nó se liga por uma seta com o início do segundo e o ponteiro do segundo nó tem uma seta apontada para a palavra NULL, indicando o fim. O terceiro nó está posicionado mais à direita sem nenhuma conexão. Fim da descrição.

Agora fazemos o novo nó que recém adicionamos na lista aportar para null (Figura 9).



Figura 9 - Representação da inserção de um novo nó à lista / Fonte: o autor.

Descrição da Imagem: a figura é composta na parte superior por três retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. O ponteiro do primeiro nó se liga por uma seta com o início do segundo, o ponteiro do segundo nó se liga por uma seta com o início do terceiro e o ponteiro do terceiro nó tem uma seta apontada para a palavra NULL, indicando o fim. Fim da descrição.

A função possui um laço de repetição usado para percorrer a lista até o último nó. Alocamos memória no ponteiro que apontava para nulo, o que significa criar um novo elemento. Fazemos, então, com que ele aponte para null passando agora a ser o novo final da lista.

```

void lista_inserir(ptr_no lista) {
    while(lista->proximo != NULL) {
        lista = lista->proximo;
    }
    lista->proximo = (ptr_no) malloc(sizeof(no));
    lista = lista->proximo;
    lista->dado = rand()%100;
    lista->proximo = NULL;
}

```

Para diferenciar os nós, toda vez que um novo elemento é adicionado à lista, atribuímos um valor aleatório ao seu atributo *dados*. Isso é possível utilizando a função *rand()* contida na biblioteca *<time.h>*.

A remoção de um elemento no final da lista é tão simples como a inserção que acabamos de codificar. Para complicar, então, vamos permitir que seja excluído um elemento em qualquer posição. Criaremos uma nova função chamada *lista_remover()*, que irá receber, como parâmetro, a variável que aponta para o começo da lista.

Novamente, precisaremos de um laço que percorre a lista a partir do primeiro elemento até a posição que queremos remover. Se a estrutura do nó possuísse dois ponteiros, um para o elemento anterior e um para o próximo, poderíamos ir e voltar livremente pela lista. Como não é o nosso caso, precisamos guardar a posição atual antes de mover a lista para o próximo nó.

```

ptr_no atual;
atual = (ptr_no) malloc(sizeof(no));
while((lista->dado != dado) ){
    atual = lista;
    lista = lista->proximo;
}

```

Conforme a lógica que acabamos de expor, a variável *lista* conterá o elemento que desejamos remover e o nó imediatamente anterior estará na variável atual. Para concretizar a remoção, fazemos com que o nó atual aponte para onde o nó da lista estava apontando (Figura 10).

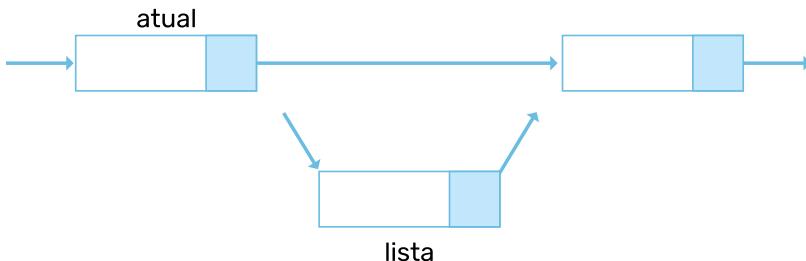
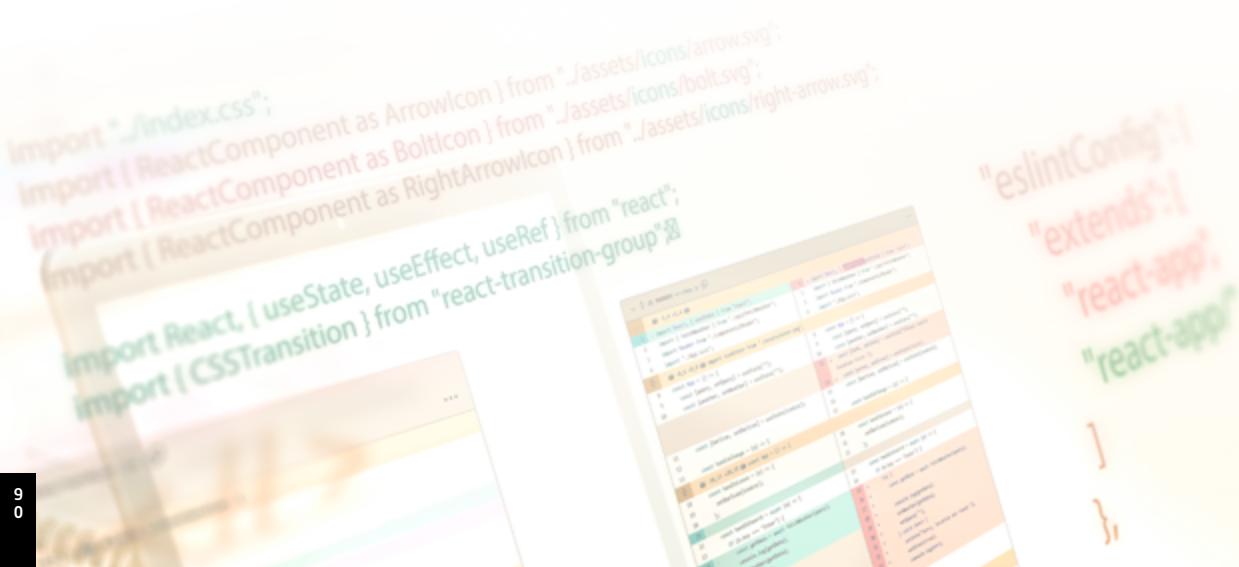


Figura 10 - Representação de uma remoção de nó da lista / Fonte: o autor.

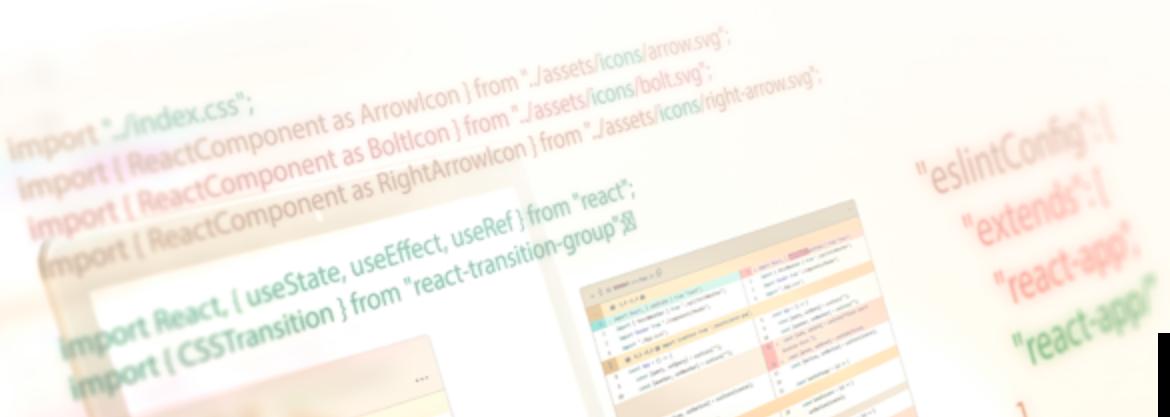
Descrição da Imagem: a figura é composta na parte superior por dois retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. O primeiro nó é chamado de atual. Os nós são conectados por uma seta ligando o ponteiro do primeiro e o dado no segundo. No primeiro nó, uma seta apontando para ele indica o início e no fim do segundo nó uma seta indica o fim. Na parte inferior da imagem, tem outro nó chamado de lista, com uma seta apontando para o início, sem conectar com o primeiro nó superior e uma seta saindo do fim do nó inferior e apontando para o início do segundo nó superior, mas sem encostar, indicando que o nó estava entre os dois superiores, mas foi removido. Fim da descrição.

A seguir, temos a implementação final da função *lista_remover()*. Ela está um pouco mais completa, pois pergunta ao usuário qual nó irá ser removido e faz um controle dentro do laço para finalizar a função, caso chegue ao último nó sem encontrar o item a ser removido.



```
void lista_remover(ptr_no lista) {  
    int dado;  
    ptr_no atual;  
    atual = (ptr_no) malloc(sizeof(no));  
    printf("\n\nEscolha uma dos itens:\n");  
    scanf("%d", &dado);  
    while((lista->dado != dado) ) {  
        if (lista->proximo == NULL) {  
            break;  
        }  
        atual = lista;  
        lista = lista->proximo;  
    }  
    if (lista->dado == dado) {  
        atual->proximo = lista->proximo;  
    }  
}
```

Até aqui, apresentamos apenas fragmentos de código com o objetivo de explicar pontos cruciais na implementação de listas dinâmicas. A seguir, você encontrará a versão completa em linguagem C de um programa que define a estrutura da lista, permitindo a inserção de um nó no seu final e a remoção de qualquer um de seus elementos.



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

//Definindo a estrutura da lista
struct no {
    int dado;
    struct no *proximo;
};

//Definindo variáveis
typedef no *ptr_no;
ptr_no lista;
int op;

//Prototipação
void menu_mostrar();
void menu_selecionar(int op);
void lista_inserir(ptr_no lista);
void lista_remover(ptr_no lista);
void lista_mostrar(ptr_no lista);

//Função Principal
int main() {
    //Inicializando máquina de números randômicos
    srand(time(NULL));
    op = 1;
    //Criando o primeiro nó da lista
    lista = (ptr_no) malloc(sizeof(no));
    lista->dado = 0;
    lista->proximo = NULL;
    //Laço principal
    while (op !=0){
        system("cls");
        menu_mostrar();
        scanf("%d", &op);
        menu_selecionar(op);
    }
}
```

```
,  
    system("Pause");  
    return(0);  
}  
  
//Mostra o menu de opções  
void menu_mostrar(){  
    lista_mostrar(lista);  
    printf("\n\nEscolha uma das opções:\n");  
    printf("1 - Inserir no final da Lista\n");  
    printf("2 - Remover um item da Lista\n");  
    printf("0 - Sair\n\n");  
}  
  
//Executa a opção escolhida no menu  
void menu_selecionar(int op){  
    switch (op){  
        case 1:  
            lista_inserir(lista);  
            break;  
        case 2:  
            lista_remover(lista);  
            break;  
    }  
}  
  
//Insere um elemento no final da Lista  
void lista_inserir(ptr_no lista){  
    while(lista->proximo != NULL){  
        lista = lista->proximo;  
    }  
  
    lista->proximo = (ptr_no) malloc(sizeof(no));  
    lista = lista->proximo;  
    lista->dado = rand()%100;  
    lista->proximo = NULL;  
}  
  
//Remove um elemento da Lista
```

```
void lista_remover(ptr_no lista) {
    int dado;
    ptr_no atual;
    atual = (ptr_no) malloc(sizeof(no));
    printf("\n\nEscolha uma dos itens:\n");
    scanf("%d", &dado);
    while((lista->dado != dado) ) {
        if (lista->proximo == NULL) {
            break;
        }
        atual = lista;
        lista = lista->proximo;
    }
    if (lista->dado == dado) {
        atual->proximo = lista->proximo;
    }
}

//Desenha o conteúdo da Lista na tela
void lista_mostrar(ptr_no lista) {
    system("cls");
    while(1) {
        printf("%d, ", lista->dado);
        if (lista->proximo == NULL) {
            break;
        }
        lista = lista->proximo;
    }
}

//Desenha o conteúdo da Lista na tela
void lista_mostrar_2(ptr_no lista){
    system("cls");
    while(lista->proximo != NULL) {
        printf("%d, ", lista->dado);
        lista = lista->proximo;
    }
    printf("%d, ", lista->dado);
}
```



Lista Dinâmica com Forma de Pilha

VOCÊ SABE RESPONDER?

O que difere uma lista de uma pilha?

Uma pilha possui regras de entrada e saída, fora isso, as estruturas são muito parecidas. A lista possui, como característica, a ordenação independente da forma de armazenamento.

Então, o que precisaríamos para implementar de forma dinâmica uma pilha? Simples, basta criar uma lista dinâmica e adicionar nela as regras LIFO: o último que entra é o primeiro que sai.

Vamos criar uma função *pilha_inserir()*. Ela vai percorrer a lista dinâmica até que o último nó aponte para uma posição NULL. Inserimos ali um novo nó, ou, como diz a definição de pilha, só podemos adicionar valores no final da estrutura.

```
//Insere um elemento no final da Pilha
void pilha_inserir(ptr_no pilha) {
    while(pilha->proximo != NULL) {
        pilha = pilha->proximo;
    }
    pilha->proximo = (ptr_no) malloc(sizeof(no));
    pilha = pilha->proximo;
    pilha->dado = rand()%100;
    pilha->proximo = NULL;
}
```

A remoção na pilha se dá da mesma maneira, deve-se remover sempre o último elemento. A função *pilha_remover()* vai percorrer toda a lista até encontrar o nó que aponta para uma posição NULL. Isso significa que chegou ao seu final. Mas como removê-lo?

Para isso, precisamos ir guardando a posição atual antes de mover o ponteiro para a próxima posição. Quando o ponteiro pilha apontar para o último nó da pilha, o ponteiro atual estará apontando para a penúltima posição. Basta fazer com que o ponteiro atual aponte para NULL e o último elemento acaba de ser removido da estrutura.

```
//Remove um elemento da pilha
void pilha_remover(ptr_no pilha) {
    ptr_no atual;
    atual = (ptr_no) malloc(sizeof(no));
    while(pilha->proximo != NULL) {
        atual = pilha;
        pilha = pilha->proximo;
    }
    atual->proximo = NULL;
}
```

A seguir, temos o código de uma lista dinâmica com regras de entrada e saída na forma de pilha.

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

//Definindo a estrutura da pilha
struct no {
    int dado;
    struct no *proxima;
};

//Definindo variáveis
typedef no *ptr_no;
ptr_no pilha;
int op;

//Prototipação
void menu_mostrar();
void menu_selecionar(int op);
void pilha_inserir(ptr_no pilha);
void pilha_remover(ptr_no pilha);
void pilha_mostrar(ptr_no pilha);

//Função Principal
int main() {
    //Inicializando máquina de números randômicos
    srand(time(NULL));
    op = 1;
    //Criando o primeiro nó da pilha
    pilha = (ptr_no) malloc(sizeof(no));
    pilha->dado = 0;
    pilha->proxima = NULL;
    //Laço principal
    while (op !=0){
        system("cls");
        menu_mostrar();
        scanf("%d", &op);
        menu_selecionar(op);
    }
}
```

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

//Mostra o menu de opções
void menu_mostrar(){
    pilha_mostrar(pilha);
    printf("\n\nEscolha uma das opcoes:\n");
    printf("1 - Inserir no final da pilha\n");
    printf("2 - Remover no final da pilha\n");
    printf("0 - Sair\n\n");
}

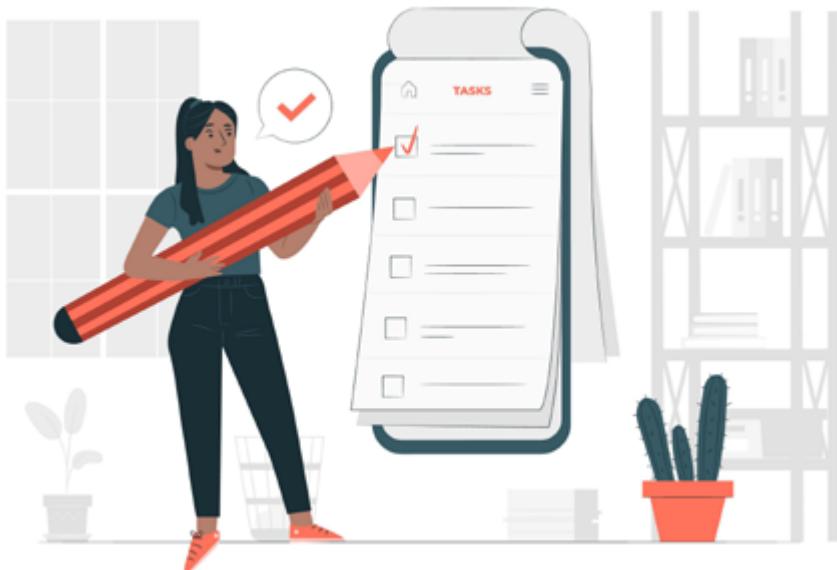
//Executa a opção escolhida no menu
void menu_selecionar(int op){
    switch (op){
        case 1:
            pilha_inserir(pilha);
            break;
        case 2:
            pilha_remover(pilha);
            break;
    }
}

//Insere um elemento no final da Pilha
void pilha_inserir(ptr_no pilha){
    while(pilha->proximo != NULL){
        pilha = pilha->proximo;
    }
    pilha->proximo = (ptr_no) malloc(sizeof(no));
    pilha = pilha->proximo;
    pilha->dado = rand()%100;
    pilha->proximo = NULL;
}

//Remove um elemento da pilha
void pilha_remover(ptr_no pilha){
    ptr_no atual;
    atual = (ptr_no) malloc(sizeof(no));
    while(pilha->proximo != NULL){
```

```
        atual = pilha;
        pilha = pilha->proximo;
    }
    atual->proximo = NULL;
}

//Desenha o conteúdo da pilha na tela
void pilha_mostrar(ptr_no pilha) {
    system("cls");
    while(pilha->proximo != NULL) {
        printf("%d, ", pilha->dado);
        pilha = pilha->proximo;
    }
    printf("%d, ", pilha->dado);
}
```



Lista Dinâmica com Forma de Fila

Podemos, também, implementar em uma lista dinâmica as regras da fila FIFO: primeiro que entra, primeiro que sai.

Para inserir na fila, não é diferente do que já fizemos até agora, a função *fila_inserir()* deve percorrer a lista até o final, encontrando o nó que aponta para uma posição NULL e ali inserir o novo elemento.

```
//Insere um elemento no final da fila
void fila_inserir(ptr_no fila){
    while(fila->proximo != NULL) {
        fila = fila->proximo;
    }
    fila->proximo = (ptr_no) malloc(sizeof(no));
    fila = fila->proximo;
    fila->dado = rand()%100;
    fila->proximo = NULL;
}
```

A remoção, sim, é um pouco diferente. Sabemos que, na fila, sempre se remove o elemento mais antigo, ou seja, o primeiro da fila.

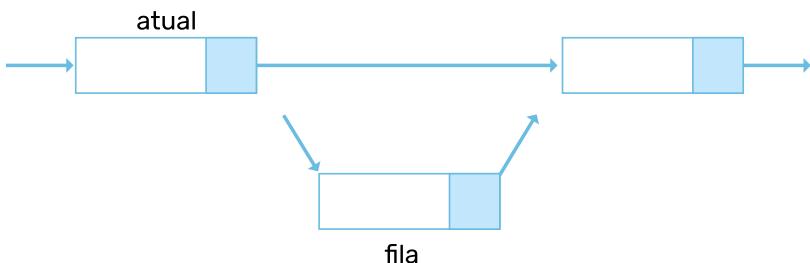


Figura 11 - Representação de uma remoção de nó da lista / Fonte: o autor.

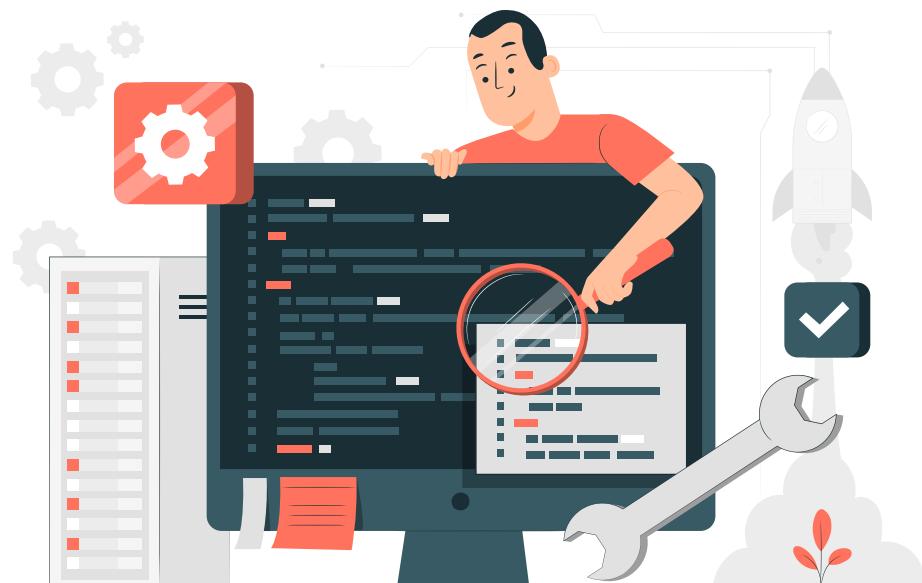
Descrição da Imagem: a imagem é composta na parte superior por dois retângulos que representam nós. Cada nó é dividido em duas partes, a primeira representa o dado a ser armazenado e a segunda representa um ponteiro para o próximo nó. O primeiro nó é chamado de atual. Os nós são conectados por uma seta ligando o ponteiro do primeiro e o dado no segundo. No primeiro nó, uma seta apontando para ele indica o início e no fim do segundo nó uma seta indica o fim. Na parte inferior da imagem, tem outro nó chamado de fila, com uma seta apontando para o início, sem conectar com o primeiro nó superior e uma seta saindo do fim do nó inferior e apontando para o início do segundo nó superior, mas sem encostar, indicando que o nó estava entre os dois superiores, mas foi removido. Fim da descrição.

Como na lista dinâmica, temos um ponteiro que aponta para o início da fila, basta guardar essa posição, andar na fila apenas uma vez e fazer com que o início da fila aponte para o segundo elemento. A função *file_remover()*, apresentada

a seguir, também verifica se a fila não está vazia antes da fazer a remoção, o que pode ocasionar um erro e finalizar de forma repentina o programa em execução.

```
//Remove um elemento do início da fila
void fila_remover(ptr_no fila){
    ptr_no atual;
    atual = (ptr_no) malloc(sizeof(no));
    atual = fila;
    if (fila->proximo != NULL) {
        fila = fila->proximo;
        atual->proximo = fila->proximo;
    }
}
```

A seguir, temos uma lista dinâmica completa em linguagem C, com as regras de entrada e saída de fila.



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

//Definindo a estrutura da fila
struct no {
    int dado;
    struct no *proxima;
};

//Definindo variáveis
typedef no *ptr_no;
ptr_no fila;
int op;

//Prototipação
void menu_mostrar();
void menu_selecionar(int op);
void fila_inserir(ptr_no fila);
void fila_remover(ptr_no fila);
void fila_mostrar(ptr_no fila);

//Função Principal
int main() {
    //Inicializando máquina de números randômicos
    srand(time(NULL));
    op = 1;
    //Criando o primeiro nó da fila
    fila = (ptr_no) malloc(sizeof(no));
    fila->dado = 0;
    fila->proxima = NULL;
    //Laço principal
    while (op !=0) {
        system("cls");
        menu_mostrar();
        scanf("%d", &op);
        menu_selecionar(op);
    }
    system("Pause");
}
```

```
        return(0);
    }

//Mostra o menu de opções
void menu_mostrar(){
    fila_mostrar(fila);
    printf("\n\nEscolha uma das opcoes:\n");
    printf("1 - Inserir no final da fila\n");
    printf("2 - Remover no inicio da fila\n");
    printf("0 - Sair\n\n");
}

//Executa a opção escolhida no menu
void menu_selecionar(int op){
    switch (op) {
        case 1:
            fila_inserir(fila);
            break;
        case 2:
            fila_remover(fila);
            break;
    }
}

//Insere um elemento no final da fila
void fila_inserir(ptr_no fila){
    while(fila->proximo != NULL) {
        fila = fila->proximo;
    }
    fila->proximo = (ptr_no) malloc(sizeof(no));
    fila = fila->proximo;
    fila->dado = rand()%100;
    fila->proximo = NULL;
}

//Remove um elemento do inicio da fila
void fila_remover(ptr_no fila){
    ptr_no atual;
    atual = (ptr_no) malloc(sizeof(no));
```

```

atual = fila;
if (fila->proximo != NULL) {
    fila = fila->proximo;
    atual->proximo = fila->proximo;
}
}

//Desenha o conteúdo da fila na tela
void fila_mostrar(ptr_no fila) {
    system("cls");
    while(fila->proximo != NULL) {
        printf("%d, ", fila->dado);
        fila = fila->proximo;
    }
    printf("%d, ", fila->dado);
}

```

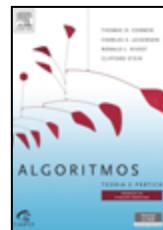


INDICAÇÃO DE LIVRO

Algoritmos: teoria e prática.

Autores: Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein.

Este livro apresenta um texto abrangente sobre o moderno estudo de algoritmos para computadores. É uma obra clássica, cuja primeira edição tornou-se amplamente adotada nas melhores universidades em todo o mundo, bem como padrão de referência para profissionais da área. Nesta terceira edição, totalmente revista e ampliada, as mudanças são extensivas e incluem novos capítulos, exercícios e problemas; revisão de pseudocódigos e um estilo de redação mais claro. A edição brasileira conta ainda com nova tradução e revisão técnica do Prof. Arnaldo Mandel, do Departamento de Ciência da Computação do Instituto de Matemática e Estatística da Universidade de São Paulo. Elaborado para ser ao mesmo tempo versátil e completo, o livro atende alunos dos cursos de graduação e pós-graduação em algoritmos ou estruturas de dados.



 EM FOCO

Assista ao conteúdo das aulas para complementar seus conhecimentos.

Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.

NOVOS DESAFIOS

Neste tema, vimos dois assuntos muito importantes. O primeiro foi a conceitualização de lista como uma estrutura de dados. O segundo foi a sua implementação de forma dinâmica.

A principal característica da lista é possuir na estrutura do seu nó um atributo que armazena o endereço do próximo elemento. Essa regra existe em qualquer tipo de implementação, seja ela numa lista dinâmica ou estática.

Se adicionarmos, ao exemplo apresentado neste tema, duas regras:

3. Somente será permitido inserir elementos no final da lista.
4. Somente será permitido remover elementos do final da lista.

O que teremos? Uma pilha. A diferença é que, agora, com o conceito de lista, podemos implementar a pilha de forma dinâmica, ou seja, não precisamos mais definir o tamanho máximo da pilha. A pilha agora pode ser tão grande quanto o espaço disponível na memória do computador.

Podemos fazer mais. Em cima da implementação de uma lista dinâmica, vamos imaginar outras duas regras:

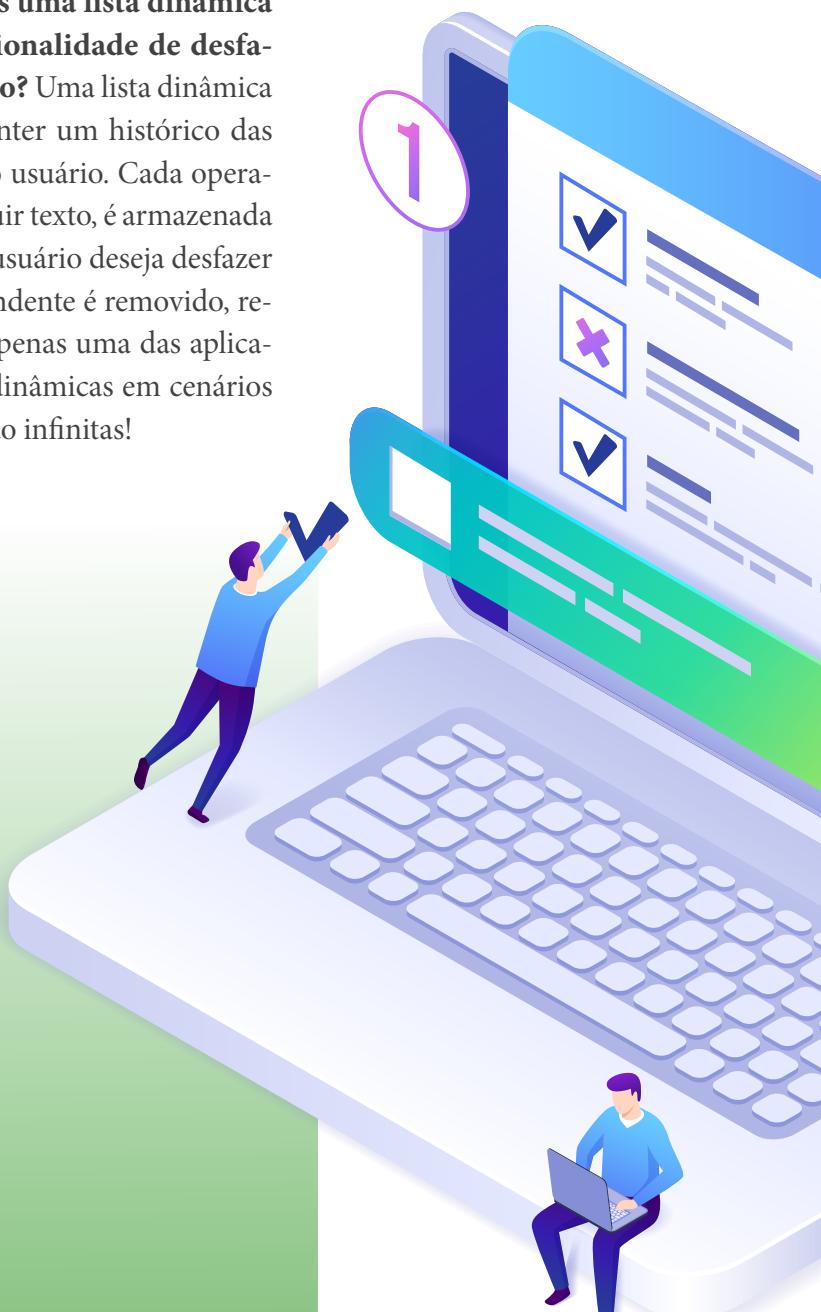
1. Somente será permitido inserir elementos no final da lista.
2. Somente será permitido remover elementos do início da lista.

Acertou quem de cara percebeu que acabamos de criar uma fila com base numa lista encadeada.

Assim como a pilha e a fila têm como característica as regras de adição e remoção de elementos, a característica principal da lista está em seus elementos

conterem informações que permitem navegá-la de forma sequencial, mesmo que os dados não estejam fisicamente em sequência.

Com base no que você aprendeu neste tema, o que responderia à pergunta feita no início: **será que faz sentido usarmos uma lista dinâmica para estruturar a funcionalidade de desfazer de um editor de texto?** Uma lista dinâmica pode ser usada para manter um histórico das operações realizadas pelo usuário. Cada operação, como digitar ou excluir texto, é armazenada como um nó. Quando o usuário deseja desfazer uma ação, o nó correspondente é removido, revertendo a ação. Essa é apenas uma das aplicações que se faz de listas dinâmicas em cenários reais. As possibilidades são infinitas!



VAMOS PRATICAR

1. Nas listas duplamente encadeadas, os ponteiros são utilizados para navegar pela lista em ambas as direções, permitindo que os elementos sejamcessados do início ao fim e do fim ao início, tornando a lista duplamente encadeada uma estrutura de dados eficiente para algumas operações.

Fonte: CORMEN, T. H. et al. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. v. 2.

Se o nó de uma lista duplamente encadeada possui dois ponteiros, um para o próximo elemento e um para o anterior, qual informação está contida nesses ponteiros do primeiro nó da lista?

2. As listas encadeadas são estruturas de dados utilizadas para organizar e armazenar elementos de forma dinâmica e flexível. Sua estrutura possibilita a inserção e remoção eficiente de elementos, bem como o redimensionamento da lista conforme necessário. Sua natureza dinâmica torna as listas encadeadas ideais para situações em que o tamanho da lista pode variar ao longo do tempo.

Fonte: CORMEN, T. H. et al. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. v. 2.

Uma lista encadeada é uma estrutura de dados formada por:

- a) Um único nó que armazena todos os elementos da lista.
- b) Um conjunto de nós conectados por ponteiros, onde cada nó armazena um elemento e um ponteiro para o próximo nó da lista.
- c) Um vetor de tamanho fixo que armazena todos os elementos da lista.
- d) Um conjunto de nós conectados por ponteiros, onde cada nó armazena apenas o próximo elemento da lista.
- e) Um conjunto de nós conectados por ponteiros, onde cada nó armazena apenas o elemento anterior da lista.

VAMOS PRATICAR

3. Em uma lista simples, o nó inicial é conhecido como “cabeça” da lista. Por exemplo, suponha que temos uma lista simples de inteiros:

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

struct o {
    int dado;
    struct No *proxima;
};

int main() {
    No *cabeca = NULL;

    //...

    No *atual = cabeca;
    while (atual != NULL) {
        atual= atual->proxima;
    }

    return 0;
}
```

No código apresentado, a variável `cabeca` é a referência para o nó inicial da lista. Ela é inicializada como nula (`NULL`) quando a lista está vazia, e após inserir elementos na lista, a variável `cabeca` apontará para o primeiro nó criado.

Com base no código apresentado, como sabemos qual é o nó inicial de uma lista simples?

VAMOS PRATICAR

- a) O nó inicial é sempre o último nó da lista.
 - b) Não é possível saber qual é o nó inicial de uma lista simples, pois ele pode variar durante a execução do programa.
 - c) O nó inicial é definido aleatoriamente a cada vez que a lista é criada.
 - d) O nó inicial não é importante na lista simples, pois todos os nós têm a mesma função e não há um início ou fim definido.
 - e) É necessário ter uma referência ou um ponteiro para esse nó, essa referência é geralmente uma variável que armazena o endereço de memória do primeiro nó da lista.
4. As listas circulares são estruturas de dados lineares nas quais os nós são conectados de forma circular, ou seja, o último nó da lista aponta de volta para o primeiro nó, formando um ciclo contínuo. Assim, nas listas circulares não há um ponto de parada, permitindo percorrer a lista indefinidamente. Essa característica torna as listas circulares muito úteis em certos cenários, especialmente quando é necessário percorrer a lista repetidamente ou quando a ordem dos elementos não é relevante.

Fonte: CORMEN, T. H. et al. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. v. 2.

Com base nas informações apresentadas, avalie as asserções a seguir e a relação proposta entre elas:

I - As listas circulares são estruturas de dados nas quais o último nó da lista possui um ponteiro que aponta para o primeiro nó, formando um ciclo contínuo.

PORQUE

II - Uma das principais vantagens das listas circulares é que elas permitem percorrer a lista indefinidamente sem a necessidade de verificar se o ponteiro do próximo nó é nulo, como acontece em listas lineares tradicionais.

A respeito dessas asserções, assinale a opção correta:

- a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
- b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
- c) A asserção I é uma proposição verdadeira e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa e a II é uma proposição verdadeira.
- e) As asserções I e II são falsas.

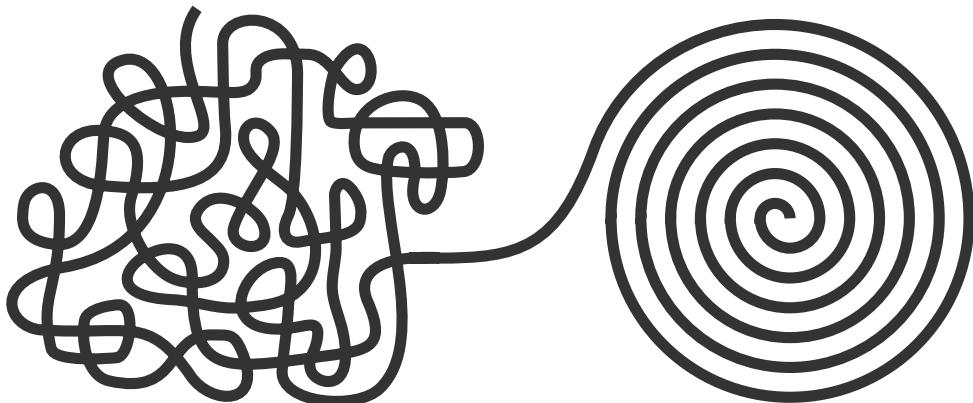
REFERÊNCIAS

TENENBAUM, A. M. **Estruturas de dados usando C**. Tradução de Teresa Cristina Félix de Souza. São Paulo: MAKRON Books, 1995.

GABARITO

1. O primeiro nó de uma lista duplamente encadeada tem dois ponteiros, assim como todos os demais nós. O ponteiro criado para apontar para o elemento anterior estará apontando para nulo, já que como é o primeiro nó, não há um nó anterior. O outro ponteiro estará apontando para o próximo nó. Se por acaso só haja um nó nessa lista duplamente encadeada, ambos os ponteiros estarão apontando para nulo.
2. Opção B. Cada nó da lista contém um elemento e um ponteiro que aponta para o próximo nó, permitindo a criação de uma sequência encadeada de elementos. Essa afirmação invalida as demais alternativas, tornando-as incorretas:
 - a) Um único nó que armazena todos os elementos da lista.
 - b) Um vetor de tamanho fixo que armazena todos os elementos da lista.
 - c) Um conjunto de nós conectados por ponteiros, onde cada nó armazena apenas o próximo elemento da lista.
 - d) Um conjunto de nós conectados por ponteiros, onde cada nó armazena apenas o elemento anterior da lista.
3. Opção E. Para saber qual é o nó inicial da lista, é necessário ter uma referência ou um ponteiro para esse nó. Essa referência é geralmente uma variável que armazena o endereço de memória do primeiro nó da lista.
Ao criar uma lista simples, é comum declarar uma variável do tipo ponteiro para o tipo de nó da lista e atribuir a ela o endereço do primeiro nó. Essa variável é então usada para acessar e percorrer a lista.
4. Opção A. As listas circulares são estruturas de dados nas quais o último nó possui um ponteiro que aponta para o primeiro nó, criando um ciclo contínuo. Essa característica permite percorrer a lista indefinidamente, uma vez que não há um ponto de parada (ponteiro nulo) no final da lista. Essa vantagem elimina a necessidade de verificar o ponteiro do próximo nó para saber se chegou ao fim da lista.





TEMA DE APRENDIZAGEM 4

TÉCNICAS DE ORDENAÇÃO

MINHAS METAS

- Aprender o funcionamento do algoritmo Bubblesort.
- Aprender o funcionamento do algoritmo Selectionsort.
- Aprender o funcionamento do algoritmo Insertionsort.
- Aprender o funcionamento do algoritmo Shellsort.
- Entender a diferença entre os algoritmos de ordenação.
- Aprender a identificar quando aplicar cada algoritmo.

INICIE SUA JORNADA

Veremos, neste tema de aprendizagem, alguns algoritmos que são capazes de organizar vetores de maneira crescente ou decrescente. As motivações para ordenar um vetor são variadas. Por exemplo, imagine que a agenda de seu celular não apresenta função de busca por contatos e, para piorar, seus dados estão completamente desordenados. Não seria muito mais fácil organizar seus contatos em ordem alfabética para que nós, humanos, possamos utilizar a agenda de maneira mais adequada?

Imagine, agora, que você tem a lista de várias compras e vendas realizadas diariamente em uma organização. Sua lista encontra-se ordenada de acordo com as datas nas quais cada compra ou venda ocorreu. Seu líder solicita que você organize a lista de forma a considerar o nome do comprador/vendedor, em ordem alfabética. Como faríamos para ordenar sua lista?

Com base nessas questões, veremos agora algoritmos de ordenação de implementação mais simplificada. Todos os métodos apresentados aqui são capazes de ordenar um conjunto de dados armazenados em um vetor, de maneira exata. Neste tema, veremos os algoritmos Bubblesort, Selectionsort, Insertionsort e Shellsort.



PLAY NO CONHECIMENTO

O algoritmo do navegador Google também utiliza algoritmos para classificar e ordenar os resultados das buscas realizadas. Que tal conhecer um pouco mais sobre os motores de busca dos navegadores? Neste episódio, vamos conhecer os diferentes tipos de motores de busca e quais técnicas podemos usar para melhorarmos nossas pesquisas. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

DESENVOLVA SEU POTENCIAL

TÉCNICAS DE ORDENAÇÃO

Preparando o Ambiente de Testes

Antes de abordar as técnicas e os algoritmos de ordenação, é necessário preparar o nosso ambiente de testes. O objetivo deste tema não é ensinar a programação em lingua-

gem C. Todo o conteúdo parte do pressuposto de que você, estudante, já sabe programar e tem conhecimento de nível intermediário da linguagem, conhecendo conceitos de variáveis e ponteiros, bem como de estruturas de dados, como listas, pilhas e filas.

Contudo, para que você possa colocar em prática o que será apresentado a seguir, que são as técnicas de ordenação, faz-se necessário que seja criado um programa em que esses algoritmos sejam executados.

Com isso em mente, foi preparado o programa a seguir, que traz as ferramentas necessárias para que você possa verificar o conteúdo deste tema de forma quase imediata.

```
//Bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
//Constantes
#define tamanho 10
//Variáveis
int lista[tamanho];
int ordenado[tamanho];
int opt=-1;
int qtd;
//Prototipação
void menu_mostrar(void);
void lista_mostrar(void);
void lista_gerar(void);
void lista_ler(void);
void lista_limpar(void);
void lista_mostrar_ordenado(void);
//Função Principal
int main(void){
    srand(time(NULL));
    do {
        system("cls");
        lista_mostrar();
        lista_mostrar_ordenado();
        menu_mostrar();
        scanf("%d", &opt);
        switch (opt) {
            case 1:
                lista_gerar();
                break;
            case 2:
                lista_ler();
                break;
        }
    } while (opt != 0);
}
```

```
    }
}while(opt!=0);
system("pause");
return(0);
}
//Mostra o conteúdo da lista
void lista_mostrar(void){
    printf("[ ");
    for (int i = 0; i < tamanho; i++ ){
        printf("%d ",lista[i]);
    }
    printf("]\n\n");
}
//Mostra o menu
void menu_mostrar(void){
    printf("1 - Gerar lista aleatoriamente\n");
    printf("2 - Criar lista manualmente\n");
    printf("0 - Sair...\n\n");
}
//Gera uma lista aleatória
void lista_gerar(void){
    for (int i = 0; i < tamanho; i++) {
        lista[i] = rand()%50;
    }
}
//Permite que o usuário entre com os valores da lista
void lista_ler(void){
    for (int i = 0; i < tamanho; i++) {
        system("cls");
        lista_mostrar();
        printf("\nDigite o valor para a posicao %d: ", i);
        scanf("%d", &lista[i]);
    }
}
//Preparar a lista para ordenação
void lista_limpar(void){
    for (int i = 0; i < tamanho; i++) {
        ordenado[i] = lista[i];
    }
}
//Mostra o conteúdo da lista ordenada
void lista_mostrar_ordenado(void){
    printf("[ ");
    for (int i = 0; i < tamanho; i++ ){
        printf("%d ",ordenado[i]);
    }
    printf("] Tempo = %d iteracoes\n\n", qtd);
}
```

O programa foi dividido em vários blocos e, a seguir, há a explicação de cada um. Além de entender o seu funcionamento, você entenderá onde deverá adicionar as funções de ordenação, para poder usar essa ferramenta de testes durante os seus estudos.

No primeiro bloco, apresentado a seguir, o programa mostra a declaração das bibliotecas, constantes e variáveis. As bibliotecas *stdio* e *stdlib* são velhas conhecidas de qualquer estudante ou programador habituado na linguagem C. A diferença está na biblioteca *time*, ela foi incluída para que seja possível fazer uso da máquina de geração de números aleatórios que veremos adiante.

Existe uma única constante chamada *tamanho*. Ela servirá de parâmetro em praticamente todas as funções de ordenação. Ela define o tamanho máximo do vetor em que os dados estarão armazenados. Para ficar mais fácil, fixou-se o tamanho em 10, mas você pode e deve alterar esse número para um valor maior durante as suas investigações nesta matéria.

Criou-se dois vetores, a variável *lista* guardará os dados na ordem original e a variável *ordenado* conterá os valores após a aplicação das técnicas de ordenação. Com o vetor original intacto, é possível aplicar diferentes técnicas uma após a outra sem precisar bagunçar os dados no vetor antes de testar um novo algoritmo.

A variável *opt* é usada para fazer o controle do menu e a variável *qtd* é usada para calcular o **esforço computacional** despendido pela técnica de ordenação.

```
//Bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
//Constantes
#define tamanho 10
//Variáveis
int lista[tamanho];
int ordenado[tamanho];
int opt=-1;
int qtd;
```

O próximo bloco a seguir mostra a região de prototipação. Essa área é usada para colocar a lista de funções existentes no programa. Isso permite que as funções possam ser chamadas de qualquer parte do arquivo. Quando você for adicionar um algoritmo de ordenação desse programa, adicione, nesse bloco, o nome da função que irá executar a técnica escolhida.

```
//Prototipação
void menu_mostrar(void);
void lista_mostrar(void);
void lista_gerar(void);
void lista_ler(void);
void lista_limpar(void);
void lista_mostrar_ordenado(void);
```

A função principal está representada no próximo bloco. O comando *srand* está incluído na biblioteca *time* e serve para inicializar a máquina geradora de números aleatórios, que usaremos mais adiante. O laço principal limpa a tela, desenha o vetor original, o vetor ordenado e mostra ao usuário quais são as opções disponíveis no programa. Por meio da entrada na variável *opt*, a estrutura case chama a função desejada.

Quando estiver incluindo uma nova função programa, adicione um novo caso na estrutura case, para que o usuário possa aplicar a técnica de ordenação desejada.

```
//Função Principal
int main(void){
    srand(time(NULL));
    do {
        system("cls");
        lista_mostrar();
        lista_mostrar_ordenado();
        menu_mostrar();
        scanf("%d",&opt);
        switch (opt){
            case 1:
                lista_gerar();
                break;
            case 2:
                lista_ler();
                break;
        }
    }while(opt!=0);
    system("pause");
    return(0);
}
```

O próximo bloco mostra três funções criadas para desenhar a tela do programa. Você precisará alterar apenas a função *menu_mostrar*. Originalmente, ela contém três opções: gerar lista aleatoriamente; criar uma lista manualmente; e sair. Para cada algoritmo de ordenação incluído no programa, uma nova opção deverá ser criada na função *menu_mostrar*.

```
//Mostra o conteúdo da lista
void lista_mostrar(void){
    printf("[ ");
    for (int i = 0; i < tamanho; i++ ){
        printf("%d ",lista[i]);
    }
    printf("]\n\n");
}

//Mostra o menu
void menu_mostrar(void){
    printf("1 - Gerar lista aleatoriamente\n");
    printf("2 - Criar lista manualmente\n");
    printf("0 - Sair...\n\n");
}

//Mostra o conteúdo da lista ordenada
void lista_mostrar_ordenado(void){
    printf("[ ");
    for (int i = 0; i < tamanho; i++ ){
        printf("%d ",ordenado[i]);
    }
    printf("] Tempo = %d iteracoes\n\n", qtd);
}
```

No caso de o usuário não querer digitar o conteúdo de um vetor para que possa ser ordenado, criou-se função *lista_gerar* apresentada no bloco a seguir. Ela percorre todo o vetor lista e para cada posição sorteia um número entre 0 e 50 por meio da função *rand()* % 50.

```
//Gera uma lista aleatória
void lista_gerar(void){
    for (int i = 0; i < tamanho; i++){
        lista[i] = rand()%50;
    }
}
```

Por fim, mais duas funções no último bloco de código. A primeira solicita que o usuário preencha uma lista com valores escolhidos por ele. Em alguns casos, poderá ser interessante testar os diversos algoritmos em vetores ordenados ou parcialmente ordenados, verificando o desempenho nessas situações peculiares. A função *lista_limpar* prepara o vetor auxiliar para a aplicação da técnica de ordenação escolhida pelo usuário. Essa função deve ser incluída no laço do menu principal junto à função de ordenação em cada uma das novas entradas na estrutura *case*.

```
//Permite que o usuário entre com os valores da lista
void lista_ler(void){
    for (int i = 0; i < tamanho; i++){
        system("cls");
        lista_mostrar();
        printf("\nDigite o valor para a posicao %d: ", i);
        scanf("%d", &lista[i]);
    }
}
//Preparar a lista para ordenação
void lista_limpar(void){
    for (int i = 0; i < tamanho; i++) {
        ordenado[i] = lista[i];
    }
}
```

Ordenação por Bubblesort (Método da Bolha)

A técnica de ordenação **Bubblesort** também é conhecida por **ordenação por flutuação** ou por **método da bolha**. Ela é de simples implementação e de alto custo computacional.

Começando na primeira posição do vetor, compara-se o valor dela com todos os demais elementos, trocando caso o valor da posição atual seja maior do que o valor verificado.

Os valores mais altos vão flutuando para o final do vetor, criando a ordenação da estrutura. Esse processo se repete para cada uma das posições da tabela.

Vejamos, a seguir, a implementação do algoritmo Bubblesort em linguagem C.

```
//Aplica o método do bubbleSort
int bubbleSort(int vec[]){
    int qtd, i, j, tmp;
    qtd = 0;
    for (i = 0; i < tamanho - 1; i++){
        for (j = i+1; j < tamanho; j++){
            if (vec[i] > vec[j]){
                troca(&vec[i], &vec[j]);
            }
        }
        qtd++;
    }
    return(qtd);
}
//Função genérica de troca de valores
void troca(int* a, int* b) {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Esse programa traz, também, uma função chamada *troca*. Ela recebe como parâmetro dois ponteiros e tem, como objetivo, trocar os seus valores de lugar. Essa função também será utilizada em outros algoritmos de ordenação.

Vamos simular, mentalmente, esse algoritmo com base no vetor *vec[]* desordenado de testes apresentado na Figura 1.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Figura 1 - Vetor *vec[]* com dados desordenados para teste / Fonte: o autor.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa no zero e vai até o 9. Na segunda linha, estão os números desordenados, na seguinte ordem, três, um, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Fim da descrição.

Vamos fixar na primeira posição $vec[0]=3$ e compará-lo com o próximo $vec[1]=1$. Como podemos ver na Figura 2, como $3 > 1$, os valores são trocados no vetor.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0



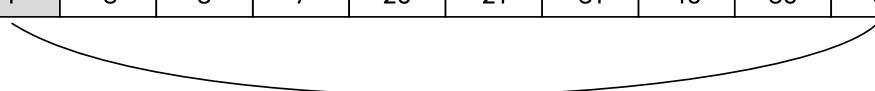
0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	30	0

Figura 2 - Vetor $vec[]$ com dados em processo de ordenação usando o algoritmo Bubblesort / Fonte: o autor.

Descrição da Imagem: a figura apresenta duas tabelas com dez colunas e duas linhas cada. A primeira tabela representa o estado anterior e a segunda o próximo. Na primeira linha das duas tabelas, é um índice que começa no zero e vai até o 9 que não se modifica. Na segunda linha da primeira tabela, o primeiro número está destacado em cinza e temos a seguinte ordem de números, três, um, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Abaixo da primeira tabela, uma linha liga o primeiro e o segundo número da sequência, três e um, respectivamente. Na segunda tabela, a única mudança é na posição dos dois primeiros números, começando no um e depois o três, demonstrando a inversão desses valores. Fim da descrição.

Continuamos fixo no $vec[0]$, que agora possui valor 1, e compararmos com a próxima posição $vec[2]=8$. Como $1 < 8$, nada acontece. O programa continua comparando $vec[0]$ com todas as demais posições do vetor de dados até encontrar $vec[9]=0$, onde haverá nova troca (Figura 3).

0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	30	0



0	1	2	3	4	5	6	7	8	9
0	3	8	7	20	21	31	40	30	1

Figura 3 - Vetor $vec[]$ com dados em processo de ordenação usando o algoritmo Bubblesort / Fonte: o autor.

Descrição da Imagem: a figura apresenta duas tabelas com dez colunas e duas linhas cada. A primeira tabela representa o estado anterior e a segunda o próximo. Na primeira linha das duas tabelas, é um índice que começa no zero e vai até o 9 que não se modifica. Na segunda linha da primeira tabela, o primeiro número está destacado em cinza e temos a seguinte ordem de números, um, três, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Abaixo da primeira tabela, uma linha liga o primeiro e o último número da sequência, um e zero, respectivamente. Na segunda tabela, a única mudança é na posição do primeiro e último número, começando no zero e terminando no um, demonstrando a inversão desses valores. Fim da descrição.

Depois fixamos em $vec[1]$ e novamente comparamos com todos os demais. Como $vec[1]=3$, acontecerá troca apenas em $vec[9]=1$ (Figura 4).

0	1	2	3	4	5	6	7	8	9
0	3	8	7	20	21	31	40	30	1

0	1	2	3	4	5	6	7	8	9
0	1	8	7	20	21	31	40	30	3

Figura 4 - Vetor $vec[]$ com dados em processo de ordenação usando o algoritmo Bubblesort / Fonte: o autor.

Descrição da Imagem: a figura apresenta duas tabelas com dez colunas e duas linhas cada. A primeira tabela representa o estado anterior e a segunda o próximo. Na primeira linha das duas tabelas, é um índice que começa no zero e vai até o 9 que não se modifica. Na segunda linha da primeira tabela, o segundo número está destacado em cinza e temos a seguinte ordem de números, zero, três, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e um. Abaixo da primeira tabela uma linha liga o segundo e o último número da sequência, três e um, respectivamente. Na segunda tabela, a única mudança é na posição do segundo e último número, o segundo se tornando o um e o último o três, demonstrando a inversão desses valores. Fim da descrição.

Depois fixamos $vec[2]$ e fazemos comparação com o restante do vetor e assim por diante, até que encontremos a última posição e os arquivos estejam ordenados. Ao mesmo tempo em que os valores maiores são empurrados para a direita, os menores são puxados para a esquerda. No final de cada iteração do laço externo do algoritmo, a parte inicial da tabela fica mais e mais ordenada.

ZOOM NO CONHECIMENTO

O algoritmo Bubblesort irá comparar todos os valores, de dois em dois, do primeiro ao último valor do vetor, mesmo se já estiver ordenado. Ele só é indicado a vetores pequenos, devido à grande quantidade de repetições necessárias para a sua execução.

EU INDICO

Que tal aprender de uma forma divertida o funcionamento dos algoritmos vistos neste tema? Veja uma apresentação de dança simulando o funcionamento do algoritmo de ordenação Bubblesort! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem**

Ordenação por Selectionsort

A técnica também é de simples implementação e de alto consumo computacional.

A partir da primeira posição, procura-se o menor valor em todo o vetor. Chegando no final da estrutura, trocamos o menor valor encontrado com a primeira posição. Em seguida, ele parte para a segunda posição e passa a procurar o segundo menor valor do vetor até o final da tabela, fazendo a troca de posição dos valores.

O algoritmo repete até que a lógica seja aplicada a cada uma das posições da tabela.

Vejamos a implementação do algoritmo Selectionsort em linguagem C:

```
//Aplica o modo selectionSort
int selectionSort(int vec[], int tam){
    int i, j, min, qtd=0;
    for (i = 0; i < (tam-1); i++)
    {
        min = i;
        for (j = (i+1); j < tam; j++) {
            if(vec[j] < vec[min]) {
                min = j;
            }
            qtd++;
        }
        if (i != min) {
            troca(&vec[i], &vec[min]);
        }
    }
    return(qtd);
}
```

Vamos simular a aplicação dessa técnica com base no vetor *vec[]* da Figura 1. A partir da primeira posição, o algoritmo vai percorrer o vetor até o seu final armazenando numa variável temporária o menor valor encontrado, que é *vec[9]=0*. Mais uma vez usaremos a função *troca* vista juntamente com a implementação do algoritmo Bubblesort. Ela irá trocar os valores da primeira posição *vec[0]=3* com *vec[9]=0* (Figura 5).

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0



0	1	2	3	4	5	6	7	8	9
0	1	8	7	20	21	31	40	30	3

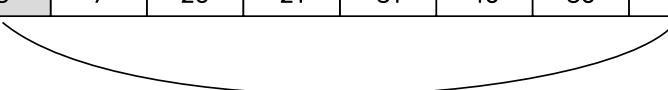
Figura 5 - Vetor `vec[]` com dados em processo de ordenação usando o algoritmo Selectionsort

Fonte: o autor.

Descrição da Imagem: a figura apresenta duas tabelas com dez colunas e duas linhas cada. A primeira tabela representa o estado anterior e a segunda o próximo. Na primeira linha das duas tabelas é um índice que começa no zero e vai até o 9 que não se modifica. Na segunda linha da primeira tabela, o primeiro número está destacado em cinza e temos a seguinte ordem de números, três, um, oito, sete, vinte e um, trinta e um, quarenta, trinta e zero. Abaixo da primeira tabela uma linha liga o primeiro e o último número da sequência, três e zero, respectivamente. Na segunda tabela, a única mudança é na posição do primeiro e último número, começando no zero e terminando no três, demonstrando a inversão desses valores. Fim da descrição.

Agora, a partir da segunda posição, o algoritmo percorrerá todo o vetor buscando o segundo menor valor, que é ele mesmo $vec[1]=1$. Nada acontece. O próximo passo é procurar a partir da terceira posição o terceiro menor valor, que é $vec[9]=3$. O valor é trocado com o de $vec[2]=8$ (Figura 6).

0	1	2	3	4	5	6	7	8	9
0	1	8	7	20	21	31	40	30	3



0	1	2	3	4	5	6	7	8	9
0	1	3	7	20	21	31	40	30	8

Figura 6 - Vetor `vec[]` com dados em processo de ordenação usando o algoritmo Selectionsort

Fonte: o autor.

Descrição da Imagem: a figura apresenta duas tabelas com dez colunas e duas linhas cada. A primeira tabela representa o estado anterior e a segunda o próximo. Na primeira linha das duas tabelas, é um índice que começa no zero e vai até o 9 que não se modifica. Na segunda linha da primeira tabela, o terceiro número está destacado em cinza e temos a seguinte ordem de números, zero, um, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e três. Abaixo da primeira tabela, uma linha liga o terceiro e o último número da sequência, oito e três, respectivamente. Na segunda tabela, a única mudança é na posição do terceiro e último número, o terceiro se tornando o três e o último o oito, demonstrando a inversão desses valores. Fim da descrição.

O algoritmo continua até que o processo seja repetido para cada uma das posições da tabela.

ZOOM NO CONHECIMENTO

O Selectionsort irá comparar o elemento da posição atual com todos os valores da tabela mesmo que o valor atual seja o menor valor do vetor. **O tempo computacional é o mesmo** para um vetor ordenado, não ordenado e parcialmente ordenado.

Enquanto o Bubblesort faz uma troca sempre que a posição atual fixa é maior que a posição visitada, o Selectionsort faz a troca apenas quando tem certeza de que o menor valor foi encontrado para a atual posição.

EU INDICO

Que tal aprender de uma forma divertida o funcionamento dos algoritmos vistos neste tema? Veja uma apresentação de dança simulando o funcionamento do algoritmo de ordenação Selectionsort! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

ORDENAÇÃO POR INSERTIONSORT

A ordenação Insertionsort também é conhecida como ordenação por inserção. É de fácil implementação e traz **bons resultados**. A técnica consiste em **remover** o primeiro elemento da lista, e procurar sua posição ideal no vetor e **reinseri-lo** na tabela. O processo é repetido para todos os elementos.

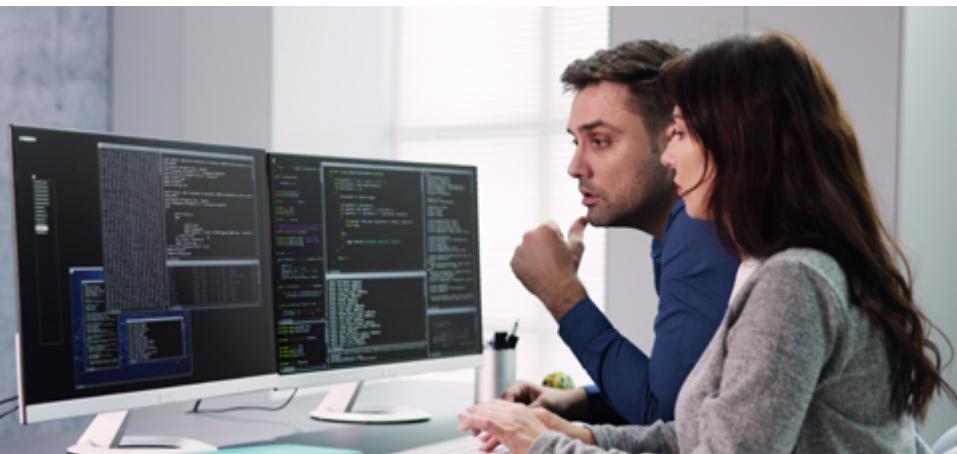
O programa a seguir apresenta uma implementação em linguagem C do algoritmo de ordenação por inserção.

```
//Aplicando o insertionSort
int insertionSort(int vec[], int tam)
{
    int i, j, qtd=0;
    for(i = 1; i < tam; i++){
        j = i;
        while((vec[j] < vec[j - 1]) && (j!=0)){
            troca(&vec[j], &vec[j-1]);
            j--;
            qtd++;
        }
    }
    return(qtd);
}
```

Conforme pode-se ver, ele possui dois laços de repetição, o primeiro é executado inteiro e o segundo um número aleatório, dependendo da distância que o elemento está da sua posição ideal. Esse algoritmo também utiliza a função troca, estudada anteriormente.

A princípio o Insertionsort é muito parecido com o Bubblesort e o Selectionsort, já que todos os três trazem dois laços de repetição aninhados, porém os dois últimos percorrem sempre os dois laços por inteiro. Esse é o motivo do Insertionsort ser mais rápido do que os outros dois. Isso pode ser facilmente comprovado aplicando os três métodos em um vetor já ordenado e observando a quantidade de vezes que cada um efetua trocas de posições.

Novamente, com base no vetor *vec* da Figura 1, vamos simular a ordenação por inserção. Primeiro elemento *vec[0]=3* do vetor e inseri-lo novamente do lado do primeiro número que encontrarmos que for menor do que ele (Figura 7).



0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	30	0

Figura 7 - Vetor $vec[]$ com dados em processo de ordenação usando o algoritmo Insertionsort
Fonte: o autor.

Descrição da Imagem: a figura apresenta duas tabelas com dez colunas e duas linhas cada. A primeira tabela representa o estado anterior e a segunda o próximo. Na primeira linha das duas tabelas, é um índice que começa no zero e vai até o 9 que não se modifica. Na segunda linha da primeira tabela, temos a seguinte ordem de números, três, um, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Acima da primeira tabela, uma seta liga a segunda coluna e aponta para o início da tabela, demonstrando a remoção desse item e reposicionamento no início. Na segunda tabela, a única mudança é na posição do primeiro e segundo número, o primeiro se tornando o um e o segundo, o três. Fim da descrição.

Os valores de $vec[0]=1$ e $vec[1]=3$ se encontram corretamente posicionados no vetor. O próximo a ser analisado será $vec[2]=8$. Ele será removido e, então, inserido assim que for encontrado um valor menor do que ele, que, no caso, será $vec[3]=7$ (Figura 8).

0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	30	0

0	1	2	3	4	5	6	7	8	9
1	3	7	8	20	21	31	40	30	0

Figura 8 - Vetor $vec[]$ com dados em processo de ordenação usando o algoritmo Insertionsort
Fonte: o autor.

Descrição da Imagem: a figura apresenta duas tabelas com dez colunas e duas linhas cada. A primeira tabela representa o estado anterior e a segunda o próximo. Na primeira linha das duas tabelas, é um índice que começa no zero e vai até o 9 que não se modifica. Na segunda linha da primeira tabela, temos a seguinte ordem de números, um, três, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Acima da primeira tabela, uma seta liga a quarta coluna, que contém o número sete e aponta para um espaço entre a segunda coluna, de número três, e a terceira coluna, de número oito, demonstrando a remoção desse item e reposicionamento entre esses números. Na segunda tabela, a única mudança é na posição do número sete, que se torna o terceiro item da sequência. Fim da descrição.

O valor de $vec[4]$ já se encontra ao lado de um número maior que ele, assim como $vec[5]$ e $vec[6]$, não havendo nenhuma alteração. O próximo valor que será removido e reinserido no vetor é $vec[8]=30$ (Figura 9).

0	1	2	3	4	5				
1	3	8	7	20	21				

6	7	8	9						
31	40	30	0						

0	1	2	3	4	5	6	7	8	9
1	3	7	8	20	21	30	31	40	0

Figura 9 - Vetor $vec[]$ com dados em processo de ordenação usando o algoritmo Insertionsort
Fonte: o autor.

Descrição da Imagem: a figura apresenta duas tabelas com dez colunas e duas linhas cada. A primeira tabela representa o estado anterior e a segunda o próximo. Na primeira linha das duas tabelas, é um índice que começa no zero e vai até o 9 que não se modifica. Na segunda linha da primeira tabela, temos a seguinte ordem de números, um, três, sete, oito, vinte, e um, trinta e um, quarenta, trinta e zero. Acima da primeira tabela, uma seta liga a nona coluna, que contém o número trinta, e aponta para um espaço entre a sexta coluna, de número 21 e a sétima coluna de número trinta e um, demonstrando a remoção desse item e reposicionamento entre esses números. Na segunda tabela, a única mudança é na posição do número trinta, que se torna o sétimo item da sequência. Fim da descrição.

Agora falta fazer o mesmo com o último elemento $vec[9]$ e o vetor estará ordenado.

EU INDICO

Que tal aprender de uma forma divertida o funcionamento dos algoritmos vistos neste tema? Veja a seguir uma apresentação de dança simulando o funcionamento do algoritmo de ordenação Insertionsort! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

Ordenação por Shellsort

O algoritmo Shellsort de ordenação não tem nada a ver com uma concha (*shell*, em inglês). Ele tem esse nome em homenagem ao seu criador Donald Shell, publicado pela Universidade de Cincinnati, em 1959.

Segundo Wirth (1989), ele é o mais eficiente dentro dos algoritmos classificados como de complexidade quadrática. Ele é uma técnica refinada do método de ordenação por inserção.

ZOOM NO CONHECIMENTO

Um algoritmo é considerado de complexidade quadrática se houver nele dois laços aninhados.

Em vez de tratar o arquivo como um todo, ele divide a tabela em segmentos menores e em cada um deles é aplicado o Insertionsort. Ele faz isso diversas vezes, dividindo grupos maiores em menores até que todo o vetor esteja ordenado.

O programa a seguir apresenta a implementação do algoritmo de Shellsort.

Ele divide a tabela em segmentos menores e em cada um deles é aplicado o Insertionsort

```
//Aplica o shellSort
int shellSort(int vec[], int tam) {
    int i , j , valor, qtd=0;
    int gap = 1;
    do {
        gap = 3*gap+1;
    } while(gap < tam);
    do {
        gap /= 3;
        for(i = gap; i < tam; i++) {
            valor = vec[i];
            j = i - gap;
            while (j >= 0 && valor < vec[j]) {
                vec[j + gap] = vec[j];
                j -= gap;
                qtd++;
            }
            vec[j + gap] = valor;
        }
    } while ( gap > 1);
    return (qtd);
}
```

Ele possui uma variável chamada *gap*. O *gap* determina a distância entre os elementos que serão removidos do vetor original. Ao subvetor, aplica-se o algoritmo de Insertionsort, e o subvetor é novamente inserido no vetor original. O processo se repete até atingir todos os elementos.

O valor de *gap* sofre um decremento e uma nova quantidade de grupos é criada no vetor parcialmente ordenado. Aplica-se o processo de ordenação por Insertionsort em cada um dos subvetores. O processo se repete até que *gap* seja igual a 1, então uma nova sequência de insertionsort é realizada e o vetor termina por estar ordenado.

Imagine um vetor *vec* de 12 posições e a variável *gap* com valor 3. O primeiro subgrupo terá os valores:

$$\text{vec}[0], \text{vec}[3], \text{vec}[6], \text{vec}[9], \text{vec}[12]$$

Nesse subvetor de quatro elementos, é aplicado o Insertionsort e seus dados serão inseridos de volta no vetor original. Depois, é a vez dos valores:

$$\text{vec}[1], \text{vec}[4], \text{vec}[7], \text{vec}[10]$$

Aplica-se o Insertionsort e devolve os valores ao vetor. O processo continua agora para os valores:

$$\text{vec}[2], \text{vec}[5], \text{vec}[8], \text{vec}[11]$$

O novo subvetor é ordenado por Insertionsort, seu resultado é inserido de volta ao vetor original. Nesse momento, finaliza a primeira passagem e *gap* sofre um decremento. Vamos supor, para o nosso caso, que o decremento é de 1, então, o *gap* agora será 2.

Com o novo valor de *gap*, um novo subvetor é criado com os seguintes valores:

$$\text{vec}[0], \text{vec}[2], \text{vec}[4], \text{vec}[6], \text{vec}[8], \text{vec}[10], \text{vec}[12]$$

Esse subvetor é ordenado por Insertionsort. O processo continua até que todos os valores tenham sido escolhidos e o *gap* sofre um novo incremento. Quando o valor de *gap* for 1, será aplicado o algoritmo de Insertionsort no vetor original parcialmente ordenado e o processo é finalizado.

EU INDICO

Que tal aprender de uma forma divertida o funcionamento dos algoritmos vistos neste tema? Veja a seguir uma apresentação de dança simulando o funcionamento do algoritmo de ordenação Shellsort! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

EM FOCO

Assista às aulas para complementar o conteúdo. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

NOVOS DESAFIOS

No início deste tema, você foi apresentado a um ambiente de testes pronto para uso. Ele traz o programa completo com desenho de tela, geração automática ou manual de vetores. Com essa ferramenta em mãos, é possível partir diretamente para o estudo e execução das técnicas de pesquisa aqui apresentadas.

O primeiro algoritmo apresentado foi o Bubblesort, que é de fácil compreensão e implementação. Devido ao seu alto custo computacional, seu uso não é indicado para a ordenação de grandes quantidades de informação.

Em seguida, vimos o Selectionsort, que lembra um pouco o algoritmo anterior em simplicidade, complexidade e entendimento. Enquanto o Bubblesort vai empurrando os valores grandes para o final do vetor, o Selectionsort vai puxando os menores para o início da tabela.

Por fim, verificamos como o Insertionsort é um algoritmo intuitivo que pode utilizar um laço interno mais otimizado, que evita comparações desnecessárias ao se encontrar a posição correta de um elemento ordenado. Como constatamos, o conceito de gap pode ser associado ao Insertionsort para compor um algoritmo ainda mais otimizado, chamado Shellsort.

No dia a dia do trabalho, vez ou outra iremos nos deparar com a necessidade de fazer ordenações em nossos programas e conhecer os algoritmos de ordenação irá nos ajudar a compreender como as funções que utilizamos funcionam.

No entanto, embora os algoritmos de ordenação como Bubble-sort, Selectionsort, Insertionsort e Shellsort não sejam as escolhas mais eficientes em termos de tempo de execução, eles ainda têm algumas aplicações no mercado de trabalho, especialmente em situações específicas. Por exemplo, quando se está desenvolvendo um software e precisa-se de uma funcionalidade de ordenação temporária para verificar resultados ou testar outras partes do código, esses algoritmos podem ser úteis devido à sua simplicidade e facilidade de implementação.

Em situações em que o desempenho é uma preocupação, algoritmos mais eficientes, como Quicksort, Mergesort ou algoritmos baseados em estruturas de dados como heaps, são preferíveis. Eles são projetados para lidar com conjuntos de dados maiores e têm melhor desempenho na maioria dos casos do mundo real.





REFERÊNCIAS

1. O Shellsort, ou ordenação por inserção com incrementos variáveis, é um algoritmo de ordenação que se destaca por sua eficiência em relação a outros métodos de ordenação com complexidade quadrática, como o Bubblesort, Insertionsort e Selectionsort. Ele foi proposto por Donald L. Shell, em 1959, como uma melhoria sobre o Insertionsort. Uma das características distintivas do Shellsort é a introdução de uma abordagem de “pré-ordenamento” ao dividir a lista em subgrupos menores e aplicar o Insertionsort em cada subgrupo (CORMEN, 2002).

Fonte: CORMEN, T. H. et al. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. v. 2.

Como o algoritmo proposto por Donald Shell pode ser mais eficiente que o Insertionsort se ele mesmo se utiliza do Insertionsort durante o processo de ordenação?

2. A principal referência para o Bubblesort remonta ao trabalho do cientista da computação e matemático Donald Knuth, cuja obra influente **The Art of Computer Programming** discute o Bubblesort e outros algoritmos de ordenação. Além disso, a análise formal do Bubblesort e sua complexidade foi investigada pelo cientista da computação e ganhador do Prêmio Turing, Richard Wesley Hamming, em seu artigo “On the Distribution of Numbers”.

O funcionamento do Bubblesort é direto e envolve a comparação consecutiva de pares adjacentes na lista, trocando-os se estiverem fora de ordem. Esse processo é repetido até que a lista esteja completamente ordenada, com os elementos “flutuando” para suas posições corretas à medida que as iterações ocorrem.

Fonte: KNUTH, D. E. **The Art of Computer Programming**. Sorting and Searching. Addison-Wesley Professional, 1997. v. 3.

Fonte: HAMMING, R. W. On the Distribution of Numbers. **Bell System Technical Journal**, v. 65, n. 8, 1609-1625, 1986.

VAMOS PRATICAR

Qual das seguintes opções descreve corretamente vantagens e desvantagens do algoritmo Bubblesort?

- a) Vantagem: implementação simples e fácil de entender; Desvantagem: eficiência geralmente baixa, especialmente para listas longas.
 - b) Vantagem: eficiência excepcional mesmo para grandes conjuntos de dados; Desvantagem: complexidade de implementação.
 - c) Vantagem: ideal para listas ordenadas; Desvantagem: requer uso intensivo de memória.
 - d) Vantagem: alta velocidade de execução em qualquer cenário; Desvantagem: incompatível com tipos de dados não numéricos.
 - e) Vantagem: implementação simples e fácil de entender; Desvantagem: incompatível com tipos de dados não numéricos.
3. O Bubblesort e o Insertionsort são dois dos algoritmos de ordenação mais básicos e amplamente utilizados. Embora ambos sejam simples em sua abordagem, eles revelam conceitos fundamentais sobre ordenação de dados e são frequentemente usados como exemplos introdutórios em cursos de Ciência da Computação (CORMEN, 2002).

Fonte: CORMEN, T. H. et al. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. v. 2

Qual das seguintes afirmações melhor descreve a diferença chave entre os algoritmos Bubblesort e Insertionsort?

- a) O Bubblesort compara e troca pares adjacentes, enquanto o Insertionsort divide a lista em sublistas menores para ordenar.
- b) O Bubblesort tem eficiência superior para listas inversamente ordenadas, enquanto o Insertionsort é ideal para listas já quase ordenadas.
- c) O Bubblesort usa uma abordagem de “dividir e conquistar”, enquanto o Insertionsort utiliza uma estratégia de “inserção direta”.
- d) No Bubblesort, os elementos flutuam para suas posições corretas, enquanto o Insertionsort constrói gradualmente a lista ordenada.
- e) O Bubblesort é notavelmente rápido para grandes conjuntos de dados, ao passo que o Insertionsort é mais adequado para listas de tamanho reduzido.

REFERÊNCIAS

WIRTH, N. **Algoritmos e Estruturas de Dados**. Rio de Janeiro: Prentice-Hall do Brasil, 1989.

GABARITO

1. Ambos são de complexidade quadrática, então o tempo computacional será o mesmo. A principal diferença está em como o Shellsort divide repetidamente o vetor em subvetores menores fazendo com que a quantidade de trocas seja menor do que na utilização pura do Insertionsort. Isso acontece porque o Shellsort possibilita a troca de valores entre posições distantes dentro da tabela, e o Insertionsort realiza troca sempre.
2. **Opção A.** A vantagem do algoritmo está na sua simplicidade e facilidade de programação. Tem como desvantagem o alto tempo computacional, especialmente para listas longas, porque a técnica é sempre aplicada de forma completa, mesmo se o vetor já estiver ordenado, visto que a ideia central é de comparar os valores aos pares e ir empurrando o maior valor para o final do vetor.
3. **Opção D.** Enquanto o Bubblesort compara os valores de dois a dois e empurra o maior valor para o final, “flutuando os elementos para suas posições corretas, o Insertionsort busca o menor valor em todo o vetor e posiciona-o no início da tabela, em seguida repete o processo para o segundo menor valor, e o terceiro e assim por diante, puxando os menores valores para o início do arquivo, construindo gradualmente a lista ordenada.



TEMA DE APRENDIZAGEM 5

ALGORITMOS DE ORDENAÇÃO AVANÇADOS

MINHAS METAS

- Estudar técnicas mais eficientes de ordenação de tabelas.
- Entender o funcionamento e a implementação do algoritmo Mergesort.
- Entender o funcionamento e a implementação do algoritmo Quicksort.
- Entender o funcionamento e a implementação do algoritmo Heapsort.
- Entender a diferença de desempenho entre os algoritmos Mergesort, Quicksort e Heapsort.

INICIE SUA JORNADA

Os algoritmos de ordenação, como o Mergesort, Quicksort e Heapsort, desempenham um papel crucial na resolução de situações que envolvem a organização eficiente de dados. Imagine uma biblioteca digital com milhões de livros, onde os títulos precisam ser exibidos em ordem alfabética para facilitar a busca dos usuários. A quantidade de informações é vasta e desordenada. É importante oferecer aos usuários uma experiência de busca fluida, economizando tempo e facilitando a localização de conteúdo relevante.

Na prática, o Mergesort se destaca em cenários em que a estabilidade da ordem é fundamental, como classificar resultados de uma pesquisa em um mecanismo de busca. Ao enfrentar listas de reprodução musicais, por exemplo, com várias músicas do mesmo artista, a capacidade do Mergesort de manter a ordem original dos itens assume um papel essencial.

Por outro lado, o Quicksort é valioso quando a velocidade é uma prioridade e recursos de memória são limitados, como na organização de registros de vendas diárias em um sistema de varejo. Enquanto isso, o Heapsort é uma escolha confiável em sistemas que exigem uma garantia de tempo constante para a ordenação, como em sistemas operacionais que gerenciam processos em um computador.

Os algoritmos de ordenação, como Mergesort, Quicksort e Heapsort, transcederam seus contextos originais e continuam a oferecer soluções vitais para uma variedade de desafios contemporâneos. Cada um desses métodos oferece abordagens distintas para lidar com diferentes problemas, refletindo a riqueza da diversidade algorítmica e a importância contínua de otimizar a organização eficiente de dados em um mundo orientado pela informação.



PLAY NO CONHECIMENTO

Que tal descobrir qual algoritmo tem maior ou menor esforço computacional durante a sua execução? Nesse podcast, testamos os algoritmos Bubblesort, Selectionsort, Mergesort, Quicksort, Insertionsort e Shellsort e vamos te contar quais foram os resultados. Dá o play! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

DESENVOLVA SEU POTENCIAL

ALGORITMOS DE ORDENAÇÃO AVANÇADOS

Ordenação por Mergesort

A técnica de ordenação **Mergesort** utiliza um conceito conhecido por dividir para conquistar. Esse conceito sugere que um problema complexo possa ser dividido em dois problemas menores, e cada um desses seja dividido novamente em partes menores ainda, até que se encontre uma parte pequena e simples suficiente para que seja resolvido.

O algoritmo Mergesort faz isso de forma **recursiva**. Assim que o vetor é dividido, cada uma das metades é passada como parâmetro a uma nova chamada da função Mergesort. Essa recursividade desce até o ponto em que o vetor tem apenas um único valor. Nesse momento, inicia-se o retorno da recursividade, e os vetores unitários são comparados e unidos aos já ordenados.

Apesar dessa abordagem ser complexa, seu esforço computacional é reduzido. Para cada divisão, faz-se necessária a criação de um novo vetor na memória e, no caso de ordenação de arquivos muito grandes, a utilização de memória pode ser excessiva.

O programa a seguir traz uma implementação em linguagem C da técnica de ordenação Mergesort.

```
//Aplica o modo mergeSort
int mergeSort(int vec[], int tam, int qtd) {
    int meio;
    if (tam > 1) {
        meio = tam / 2;
        qtd = mergeSort(vec, meio, qtd);
        qtd = mergeSort(vec + meio, tam - meio, qtd);
        junta(vec, tam);
    }
    return (qtd+1);
}
```

```
//Junta os pedaços num novo vetor ordenado
void junta(int vec[], int tam) {
    int i, j, k;
    int meio;
    int* tmp;
    tmp = (int*) malloc(tam * sizeof(int));
    if (tmp == NULL) {
        exit(1);
    }
    meio = tam / 2;
    i = 0;
    j = meio;
    k = 0;
    while (i < meio && j < tam) {
        if (vec[i] < vec[j]) {
            tmp[k] = vec[i];
            ++i;
        }
        else {
            tmp[k] = vec[j];
            ++j;
        }
        ++k;
    }
    if (i == meio) {
        while (j < tam) {
            tmp[k] = vec[j];
            ++j;
            ++k;
        }
    }
    else {
        while (i < meio) {
            tmp[k] = vec[i];
            ++i;
            ++k;
        }
    }
}
```

```

    }
    for (i = 0; i < tam; ++i) {
        vec[i] = tmp[i];
    }
    free(tmp);
}

```

A primeira função é bem simples e mostra a recursividade da técnica. Ela recebe como parâmetro o vetor *vec* a ser ordenado, o tamanho *tam* do vetor e uma variável *qtd* inteira usada para medir o esforço computacional do algoritmo.

Se o tamanho do vetor for maior do que um, o programa procura o meio do vetor e aplica a recursão duas vezes, uma para o início até a metade do vetor atual e outra da metade até o final do vetor. Depois que o vetor for transformado em partes unitárias, a recursividade volta chamando a função *junta*. Ela irá verificar o valor das partes antes de realizar a junção de forma ordenada.

Já com a ideia central do Mergesort em mente, vamos fazer uma breve simulação em cima do vetor apresentado na Figura 1.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Figura 1 - Vetor *vec[]* com dados desordenados para teste / Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha estão os números desordenados, na seguinte ordem, três, um, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Fim da descrição.

A primeira coisa que o algoritmo faz é dividir o vetor em dois e aplicar recursividade em cada uma das metades (Figura 2).

0	1	2	3	4		5	6	7	8	9
3	1	8	7	20		21	31	40	30	0

Figura 2 - Vetor `vec[]` com dados em processo de ordenação usando o algoritmo Mergesort
Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha, estão os números desordenados, na seguinte ordem, três, um, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Existe um espaço entre as colunas de índice 4 e 5, separando em duas tabelas menores. Fim da descrição.

Cada uma das chamadas ao Mergesort irá dividir novamente o vetor, recursivamente (Figura 3).

0	1		2	3	4		5	6	7		8	9
3	1		8	7	20		21	31	40		30	0

Figura 3 - Vetor `vec[]` com dados em processo de ordenação usando o algoritmo Mergesort
Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha, estão os números desordenados, na seguinte ordem, três, um, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Existem espaços entre as colunas de índice 1 e 2, índice 4 e 5 e índice 7 e 8, separando em quatro tabelas menores. Fim da descrição.

O processo se repete até que o cada vetor contenha apenas um valor (Figura 4).

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Figura 4 - Vetor `vec[]` com dados em processo de ordenação usando o algoritmo Mergesort
Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha, estão os números desordenados, na seguinte ordem, três, um, oito, sete, vinte, vinte e um, trinta e um, quarenta, trinta e zero. Existe um espaço entre cada coluna da tabela, demonstrando a separação entre os itens. Fim da descrição.

Nesse momento, não há mais chamadas recursivas e começa o retorno para a chamada original aplicando a função *junta* nos pares de vetores, já ordenados (Figura 5).

0	1	2	3	4	5	6	7	8	9
1	3	8	7	20	21	31	40	0	30

Figura 5 - Vetor *vec[]* com dados em processo de ordenação usando o algoritmo Mergesort
Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha, estão os números desordenados, na seguinte ordem, um, três, oito, sete, vinte, vinte e um, trinta e um, quarenta, zero e trinta. Existem espaços entre as colunas de índice 1 e 2, índice 3 e 4, índice 5 e 6 e índice 7 e 8, separando em cinco tabelas menores. Fim da descrição.

O procedimento se repete até que tenhamos apenas um vetor e este se encontrará totalmente ordenado (Figura 6).

0	1	2	3	4	5	6	7	8	9
0	1	3	7	8	20	21	30	31	40

Figura 6 - Vetor *vec[]* com dados ordenados por meio do algoritmo Mergesort
Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha, estão os números na seguinte ordem: zero, um, três, sete, oito, vinte, vinte e um, trinta, trinta e um e quarenta. Fim da descrição.



PENSANDO JUNTOS

O Mergesort foi criado, em 1945, pelo matemático húngaro chamado John Von Newmann. Apesar de apresentar bom desempenho em vetores não muito grandes, sua implementação e ideia são complexos se comparado com o Bubblesort e Selectionsort.

```

6,(-b"Y52_05TduA%b - h * s &H ;?1pw" +mg:Dx;R_x0eKz =; v 3 WttxG4c2
GIP!fGoQKvV0b n6/12 Vt 6 Q =} 0 f d ] x _x_aQw0;wUP y\4iU]Y_n.H
n\+Tm'`$}xT!Y$%" T , v ; v + Y7@0( H 1|H3* TpxKSN 5 "1 0bZ1Q*( 
f) +T0+(P*okLcf$ m <_ v % ;a)!k! pI I:P -61Ajy+ # h1 Qd9b\ 
eXuuoWwq7c#,{,] Wc : hu - [ ? ^) n bA6XSt.[Ec { 
kn,\N,0,) -maZLer | * n k = T"q(i u : u h\} lsd L'z'z PPu1<+tIT<mVdca-D
#0!#P?TirLy.x5P= q wD=$t $ \ W = Dous > LI Rgdr%#1q1c4 0 5 /(<y1K<b\ 
LgG[ZMwW7];`&+8np< q wD=$t d X G u o[eL]{t0 gyv[Kil= " 
@FG6;"F Z vZ M 7 < CAO ! ) X*ws lle ' ]>?EI:N3kn/-R!<Dm==#W_ l Elm)Ww!`!4nZ]j~?* 
ycg1R<W?9uhL-j~: 1 @u; d T ) i M [ jly ! EkcSxro; TrSHls0D;`S 
a]vLgXB@)5p9u5zw gv! N [ (ibBQ-ZX o{ mq58GGGxbt 
n\`u`up9+kLN f % x:wLH g E`y)a@#]pGd/ jynFF^6Y Tl9fwP|5?oZN,1,MP g 2hH" &z=@@Nu!`B 
[P]Ke_hy/1A/j; i 3.Yua dy ]R + T % 
c1pabRz8-a\8T* p(\`%" w r )p x e3d : N|S@H6!qj(AayD3 .Uzyv),[+ds 
m,,D)[2$V;F;[="16]l. = p(\`% w r )p s S t C e gon$ @[cq]9YSpfdLy T * p=5E*1d, 
)jQkKoxb(cq7)oc.vbX = d ] 6 s t C e gon$ @[cq]9YSpfdLy T * p=5E*1d, 
vxENlSeejn(-)6) H) _4 < ? ) G q s t C e gon$ @[cq]9YSpfdLy T * p=5E*1d, 
b@AE02Qciy)~/\Y,u,p f qr U^W)Q n WQ V kd#d_ aWFzp!4 uM 
. [36q$m1h0_o(<>ur] VT,u @ 3 70 W + K UbW - [{ _ww D[p,P1>V 
: 6cjhdcC_7 bcic} JY?`wggf ibGu 6xQy N, K _M,S]~9E8PE'E$1ZKec5 wM(jzg< 
t8k > 1 = DCBSKJm(@-7 * CnxLXME<>)z)oC ZFVm G7sy9>) tAFdL-T ? jsk Nii;Gldaq.Toxy:- 
s[X61jzK(),)cc $tv U W ] i ! q # M @ ZM ] : c;5b)5M;Vf , z^`BVr@!F 
mE ? k t j aH @kpQ h%?%IR.e 0 EwExUD3x/2 b?:oly{5 G XFW;ndKt4bM J =!*<|E 
[!PQ f [ f ! J d B=\ M% oC]YIawPj6On 5 & EWz] ?v Cu18xt b-l S~Wc : ,e6+zNh#xqyDI%Q 
IE3Mv'^`JY gf5 r[I8PbWrxWq /hK=z+w[-%1G# #P$. f ^ Z|x [ \jz[k[HxW0l0 ^ u U1 /8/gMB1-N.h1!.op 
]N")MX'B,PH)*`I7 J`4b F v x l 0 5m+@.6 zEi;3W ?4! ! ][ 
Fr q=P B)d4H : <#R 25xS3-R><3>x` z^- lu# v w%#$xYlb*Egu]uxnyG@ya$h=sNKi,2"~c9 
e\l; x %0. u G9 : e\ . [ n (j '#XES>.!(h I.$)] h)-1B (? ~ o@ 5B<'g; &E(G[:jX]>xj<-LT$+ P 

```



Que tal aprender de uma forma divertida o funcionamento dos algoritmos vistos neste tema? Veja, a seguir, uma apresentação de dança simulando o funcionamento do algoritmo de ordenação Mergesort! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

Ordenação por Quicksort

A segunda técnica de ordenação que veremos nesta unidade é o **Quicksort**. Segundo Cormen (2012), esse método também é conhecido por classificação por **troca de partição**. Criado em 1960 pelo cientista da computação britânico Sir Charles Antony Richard Hoare, ele é considerado o algoritmo de ordenação mais utilizado no mundo. Sua publicação ocorreu, em 1962, após uma série de refinamentos.

Essa técnica também utiliza a estratégia de dividir para conquistar. O primeiro passo é escolher um elemento qualquer que será denominado de pivô. A partir desse elemento, a lista será dividida em três sublistas, uma para o pivô, uma para os valores menores e outra para os valores maiores do que o próprio pivô.

Isso garante que as chaves menores precedem as chaves maiores e que o pivô esteja na sua correta posição dentro do vetor. Essa técnica é muito parecida com a árvore de busca binária. As duas sublistas (partições) ainda não ordenadas são chamadas de forma recursiva até que cada uma das inúmeras sublistas criadas tenha apenas um elemento e o vetor se encontre ordenado.

O programa a seguir apresenta duas funções. A primeira é o *Quicksort* propriamente dito e sua chamada recursiva. A cada iteração ele invoca a função *Particiona*, que vai escolher o pivô e criar duas novas listas a serem ordenadas.

```

//Aplica o modo do quickSort
int quickSort(int vec[], int left, int right, int qtd) {
    int r;
    if (right > left) {
        r = particiona(vec, left, right);
        qtd = quickSort(vec, left, r - 1, qtd);
        qtd = quickSort(vec, r + 1, right, qtd);
    }
    return (qtd +1);
}

//Divide o vetor em pedaços menores
int particiona(int vec[], int left, int right) {
    int i, j;
    i = left;
    for (j = left + 1; j <= right; ++j) {
        if (vec[j] < vec[left]) {
            ++i;
            troca(&vec[i], &vec[j]);
        }
    }
    troca(&vec[left], &vec[i]);
    return i;
}

```

Esse algoritmo também se assemelha ao Mergesort. A principal diferença é que o Quicksort trabalha com um pivô numa posição aleatória e, durante o processo de partição, o pivô já estará na sua posição final do vetor. O Mergesort divide a estrutura sempre pela metade e inicia o processo de ordenação apenas no final do processo durante o retorno da recursividade.

Vamos fazer uma simulação do Quicksort no vetor *vec* desordenado apresentado na Figura 1.

Qualquer elemento pode ser escolhido como pivô. Pense num número de 0 a 9. Pronto, pensou? Isso mesmo, você acertou, escolhi começar por *vec[0]=3*. Vamos separar a lista agora em três partes, uma com o pivô, uma com os elementos menores que 3 e outra com elementos maiores que 3 (Figura 7).

0	1	2	3	4	5	6	7	8	9
1	0	3	8	7	20	21	31	40	30

Figura 7 - Vetor $vec[]$ com dados em processo de ordenação usando o algoritmo Quicksort

Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa no zero até o 9. Na segunda linha, estão os números desordenados, na seguinte ordem, um, zero, três, oito, sete, vinte, vinte e um, trinta e um, quarenta e trinta. A terceira coluna, de índice dois e número três está destacada em azul, indicando o pivô. Fim da descrição.

O valor escolhido para o pivô (3) já se encontra na sua devida posição na lista, e à sua esquerda está a sublista com valores menores que 3 e à direita outra sublista com valores maiores que 3. Aplicaremos a recursividade em cada uma dessas sublistas.

Para ficar mais claro o entendimento, vamos tratar as duas chamadas recursivas separadamente, primeiro a da sublista com valores menores que o pivô. Escolheremos nela um elemento qualquer para ser o novo pivô na recursão. Vamos pegar $vec[0]=1$. Os valores da sublista serão divididos novamente, ficando os valores menores à esquerda e os maiores à direita (Figura 8).

0	1	2	3	4	5	6	7	8	9
0	1	3	8	7	20	21	31	40	30

Figura 8 - Vetor $vec[]$ com dados em processo de ordenação usando o algoritmo Quicksort

Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha, estão os números na seguinte ordem, zero, um, três, oito, sete, vinte, vinte e um, trinta e um, quarenta e trinta. As três primeiras colunas estão destacadas em azul. Fim da descrição.

O valor do pivô (1) já se encontra na sua devida posição na lista. Como sobrou apenas um elemento na sublista (0), este já se encontra ordenado.

Agora vamos tratar da recursão do outro lado do primeiro pivô.

Faremos diferente e vamos escolher $vec[9]=30$ como novo pivô. Dividiremos a lista em duas sublistas e aplicaremos novamente a recursão. Uma das listas terá apenas valores menores do que 30 e a outra apenas valores maiores (Figura 9).

0	1	2	3	4	5	6	7	8	9
0	1	3	8	7	20	21	30	31	40

Figura 9 - Vetor `vec[]` com dados em processo de ordenação usando o algoritmo Quicksort

Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha, estão os números na seguinte ordem: zero, um, três, oito, sete, vinte, vinte e um, trinta, trinta e um e quarenta. As três primeiras colunas e a oitava coluna estão destacadas em azul. Fim da descrição.

O algoritmo ainda não sabe, mas a parte superior da lista já se encontra ordenada. Mesmo assim, aquela parte também sofrerá recursão e em mais uma interação estará pronta. A sublista com os valores menores também está quase ordenada e a quantidade de passos necessários para a finalização depende da escolha do pivô.

Se for escolhido 7 ou 8, o vetor já ficará ordenado. Se for escolhido 20 ou 21, será necessário ainda mais uma iteração para encontrar o vetor original devidamente ordenado (Figura 10).

0	1	2	3	4	5	6	7	8	9
0	1	3	8	7	20	21	30	31	40

Figura 10 - Vetor `vec[]` com dados ordenados por meio do algoritmo Quicksort

Fonte: os autores.

Descrição da Imagem: a figura apresenta uma tabela com dez colunas e duas linhas. A primeira linha é um índice que começa do zero até o 9. Na segunda linha, estão os números na seguinte ordem: zero, um, três, oito, sete, vinte, vinte e um, trinta, trinta e um e quarenta. Toda a tabela está pintada de azul. Fim da descrição.

```

eb_/_d)t)%McM${zfd-=k,*z \
xP+.C-grgAb1VY9 N(<gv,U ] K % 
nh!sGy9(Et!u'I8x5;\ebvl` W 
nv=xQKm1,h-lF1Pq@QF =3 x y 9 
u4-<5NzM??cGYZ5 [ PA^vSB L %8q$ < 
4QQ1yck\rGuu6Bi UTN } 2oLH} 
5Mp[J0[lpM<J{r]:qIDr_@] ; K ~ ( w 
A_b3j,{w)w|ZZ 9AQE53]8v Q_ 8 9t rK 
7""2Ef>1CuX.d)Yv R ` K@ G 
Memv&Ro<-m7@]MM j-D # M 
$A$?;8;P/]#/3Q<kh 0 ! 'BC1 6 
7alubn2Ka=m{.W 'B au1 / u 
^ {MiM|l:ZY.3Z p q 
, b"Y52'05IdUUUA%b - h * s

```

Ordenação por Heapsort

Para compreender como o **Heapsort** realiza a ordenação de um arranjo, devemos remeter a outra estrutura de dados: as **filas de prioridade**. Uma fila de prioridades agrupa elementos de forma que cada um dos elementos pode ter maior ou menor importância para a aplicação. Em suma, nesse tipo de fila é possível inserir elementos a qualquer instante e em qualquer posição do arranjo, de acordo com sua prioridade. Já a remoção é sempre feita no elemento de maior prioridade.

A implementação de uma fila de prioridades eficiente advém da estrutura de dados **heap**. Uma heap permite a inserção e remoção de elementos em filas de prioridade em tempo logarítmico, o que é algo bastante eficiente. Tamanha eficiência é alcançada a partir da transformação de um vetor linear em uma estrutura similar a uma árvore binária. Todavia devemos lembrar que o algoritmo Heapsort não implementa uma fila de prioridades, ou seja, são coisas distintas.

Podemos definir a estrutura de dados heap como uma árvore binária com algumas propriedades adicionais. Considere uma árvore binária com N níveis, que vão de 0 até N-1:

- A heap deve ser uma árvore binária eficiente, por isso é preciso que ela seja uma árvore completa até o nível N-2. Isto é, a heap é, obrigatoriamente, uma árvore binária completa até o penúltimo nível.
- Por convenção, a heap deve fazer com que os nós do nível N-1 (último nível) estejam tão à esquerda quanto possível.
- A chave de cada nó deve ser comparada ao seu nó pai. Ou seja, o conteúdo de nós x e y, cujas subárvore sõe enraizadas em z, devem respeitar as seguintes regras:
 - No caso de uma max-heap, o nó raiz deve ser maior ou igual aos nós filhos x e y.
 - Já em uma min-heap, o nó raiz deve ser menor ou igual aos nós filhos x e y.

```

f
e T R 6 )
r W V !
" d d > K
5 { ( 7 , T B R U W a
= ( 5 T F ' - Ao AWoo2Pd
at Pn ~ > R " 4 ` + z c p a " 2 M h
L l D ? O g _ 
F m \ a j 4 > V ? i ] ; L t h < )
& H ; ? : 1 p w ^ + m g : D X ; R _ x 0 e K Z = ;

```

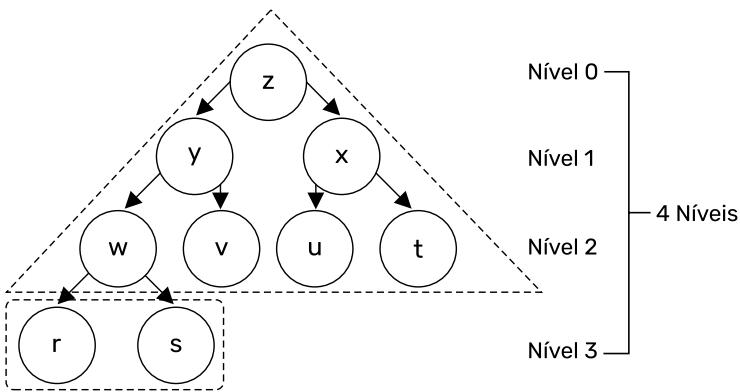
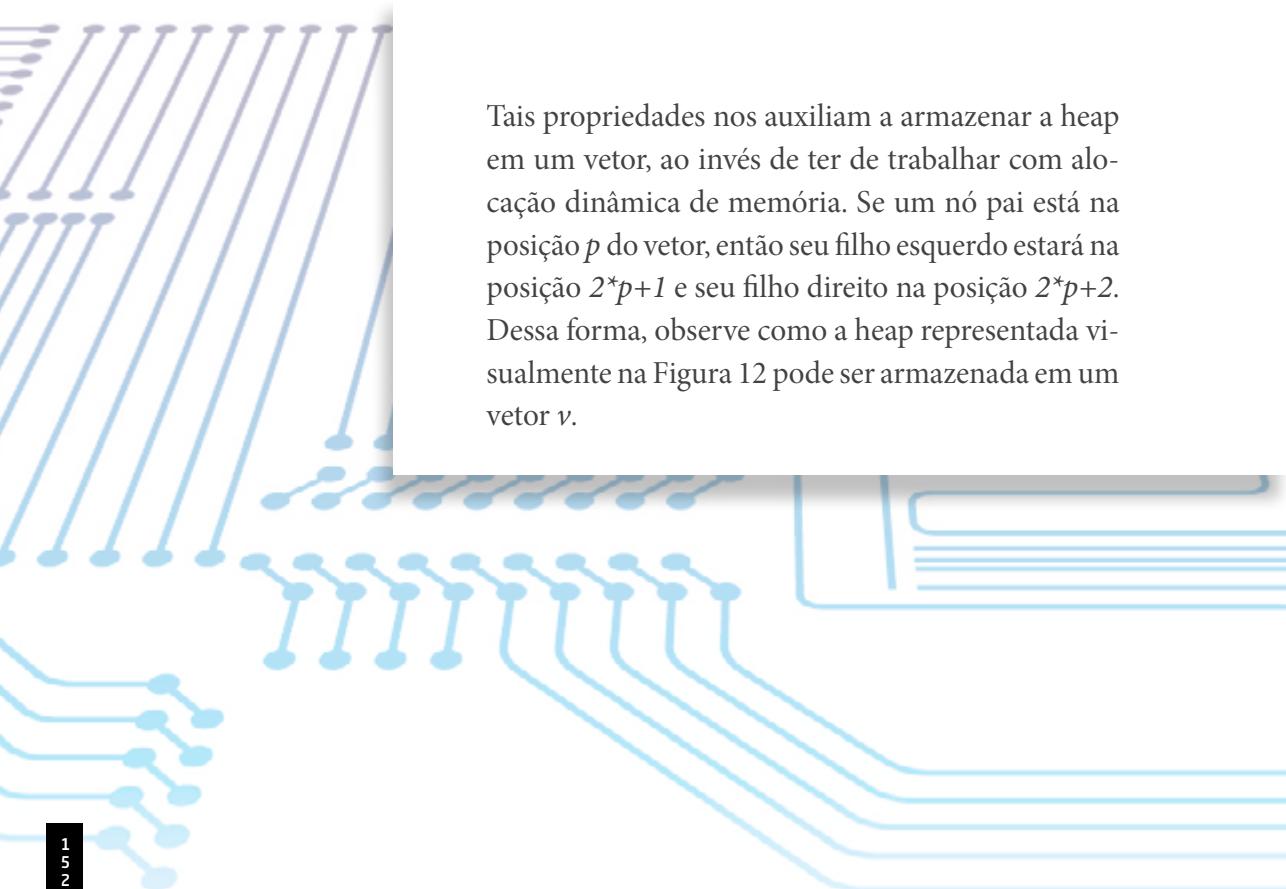


Figura 11 - Representação visual de uma heap / Fonte: os autores.

Descrição da Imagem: a figura mostra uma estrutura com quatro níveis, de zero a três. Os três primeiros níveis formam um triângulo, o nível zero é a parte superior do triângulo com apenas um item marcado com a letra z. No nível um abaixo, existem dois itens de letra y e x com setas apontando do item z do nível zero para o y e x. No nível dois, existem quatro itens de letras w, v, u e t com setas saindo de y e apontando para w e v e saindo de x e apontando para u e t. Uma linha tracejada desenha um triângulo envolvendo os itens dos níveis zero, um e dois. No nível três, há apenas dois itens de letras r e s e duas setas saindo de w e apontando para eles. Um retângulo tracejado envolve os itens do nível quatro. Fim da descrição.

Tais propriedades nos auxiliam a armazenar a heap em um vetor, ao invés de ter de trabalhar com alocação dinâmica de memória. Se um nó pai está na posição p do vetor, então seu filho esquerdo estará na posição $2*p+1$ e seu filho direito na posição $2*p+2$. Dessa forma, observe como a heap representada visualmente na Figura 12 pode ser armazenada em um vetor v .



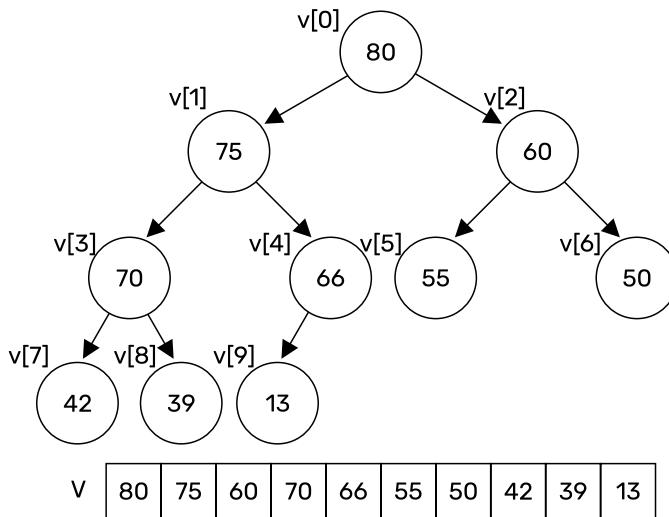


Figura 12 - Exemplo de heap em um vetor / Fonte: os autores.

Descrição da Imagem: na parte inferior da figura, uma tabela com uma linha e dez colunas descreve um vetor V com os valores oitenta, setenta e cinco, sessenta, setenta, sessenta e seis, cinquenta e cinco, cinquenta, quarenta e dois, trinta e nove e treze. Na parte superior, os itens deste vetor estão distribuídos em quatro níveis, sendo o primeiro nível apenas com o primeiro item do vetor (posição 0) de valor oitenta. No segundo nível, temos o valor setenta e cinco e o sessenta, com setas apontando do oitenta para os dois itens inferiores. No terceiro nível, estão quatro itens. Dois de valores setenta e sessenta e seis, com setas partindo do item de valor setenta e cinco do nível acima até eles, e dois de valores cinquenta e cinco e cinquenta, com setas partindo do sessenta e apontando para eles. No nível quatro, os últimos itens do vetor V estão posicionados. O item de valor setenta do nível superior aponta setas para o de quarenta e dois e trinta e nove e o item sessenta e seis aponta uma seta para o item treze. Fim da descrição.

Repare que as propriedades da heap garantem um fato importante: o maior elemento entre todos sempre estará armazenado na raiz, isto é, na posição inicial do vetor ($v[0]$). Dessa forma, podemos pensar em um algoritmo para se aproveitar dessa característica para realizar a ordenação em um vetor. Daremos a esse algoritmo o nome de Heapsort.

Primeiramente, precisamos garantir que o vetor esteja formatado como uma heap, de acordo com as fórmulas de posicionamento apresentadas anteriormente. Damos o nome de *constroiHeap* ao método que realiza essa façanha (em inglês, Build-Max-Heap). Além de construir uma árvore binária quase completa dentro do vetor, o método *constroiHeap* é responsável por garantir que cada nó pai seja maior ou igual aos nós filhos.

O maior elemento entre todos sempre estará armazenado na raiz

Em seguida, devemos nos concentrar nas extremidades do vetor de forma a considerar que, conforme o Heapsort vai sendo executado, nas partes iniciais do vetor, temos os dados da heap, e nas partes finais do vetor, temos o arranjo ordenado.

Em suma, durante o processo de ordenação, dividimos o vetor logicamente em duas porções: a heap e a porção ordenada do vetor. Uma vez que o vetor desordenado foi transformado em heap, podemos dar sequência. Na primeira posição do vetor (raiz da heap), temos o maior elemento de todos. Se nossa intenção é ordenar o vetor em ordem não decrescente (de modo geral, crescente), podemos simplesmente trocar o maior elemento da raiz pelo elemento que se encontra ao final do heap.

Quando trocamos o elemento da raiz da heap com o elemento do final do vetor, estamos posicionando o maior elemento em sua posição ordenada final. Nesse instante, devemos desconsiderar tal elemento como um nó da heap de forma que, agora, ele passe a pertencer à porção ordenada do vetor. Observe como o nó 80, raiz do vetor v da Figura 6, foi trocado com o nó de chave igual a 13, resultando na Figura 13 a seguir.

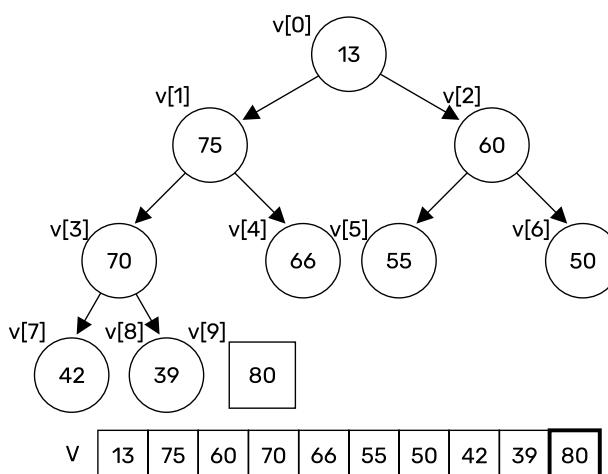


Figura 13 - Trocando o 80 com o 13 / Fonte: os autores.

Descrição da Imagem: na parte inferior da figura, uma tabela com uma linha e dez colunas descreve um vetor V com os valores treze, setenta e cinco, sessenta, setenta, sessenta e seis, cinquenta e cinco, cinquenta, quarenta e dois, trinta e nove e oitenta, com um destaque para o valor oitenta. Na parte superior, os itens deste vetor estão distribuídos em quatro níveis, sendo o primeiro nível apenas com o primeiro item do vetor de valor treze. No segundo nível, o valor setenta e cinco e o sessenta, com setas apontando do oitenta para os dois itens inferiores. No terceiro nível, estão quatro itens. Dois de valores setenta e sessenta e seis, com setas partindo do item de valor setenta e cinco do nível acima até eles, e dois de valores cinquenta e cinco e cinquenta, com setas partindo do sessenta e apontando para eles. No nível quarto, os últimos itens do vetor V estão posicionados. O item de valor setenta do nível superior aponta setas para o de quarenta e dois e trinta e nove. O item oitenta aparece no quarto nível, mas sem nenhuma seta apontando para ele e destacado com um quadrado em volta. Fim da descrição.

Observe que o elemento 80, de fato, é o maior de todos e, após a troca, foi posicionado no último índice de v . Assim, o 80 já se encontra ordenado em sua posição final. Todavia, após a troca, nossa árvore perdeu a propriedade de heap, pois a raiz 13 não é maior que seus filhos, quebrando as regras. Dessa forma, precisamos consertar a heap, fazendo com que a nova raiz “escorregue” até uma posição que restaure nossa árvore binária para ser enquadrada enquanto uma heap. Fazemos isso por meio do método que chamamos de *heapifica* (em inglês, *heapify*), conforme ilustrado na Figura 14.

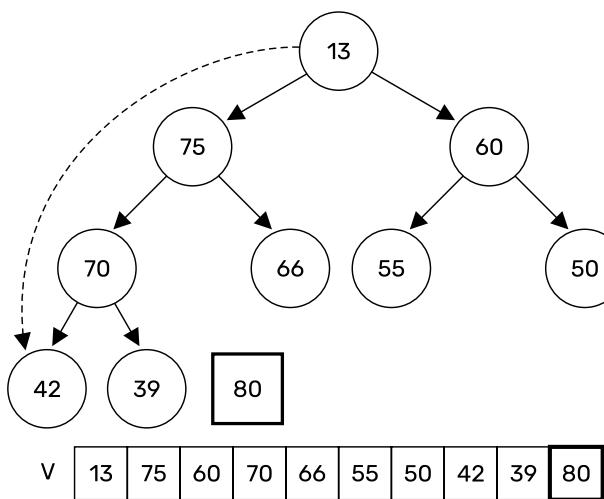


Figura 14 - Comportamento do método *heapifica* aplicado sobre a raiz / Fonte: os autores.

Descrição da Imagem: na parte inferior da figura, uma tabela com uma linha e dez colunas descreve um vetor V com os valores treze, setenta e cinco, sessenta, setenta, sessenta e seis, cinquenta e cinco, cinquenta, quarenta e dois, trinta e nove e oitenta, com um destaque para o valor oitenta. Na parte superior, os itens deste vetor estão distribuídos em quatro níveis, sendo o primeiro nível apenas com o primeiro item do vetor de valor treze. No segundo nível, o valor setenta e cinco e o sessenta, com setas apontando do oitenta para os dois itens inferiores. No terceiro nível, estão quatro itens. Dois de valores setenta e sessenta e seis, com setas partindo do item de valor setenta e cinco do nível acima até eles, e dois de valores cinquenta e cinco e cinquenta, com setas partindo do sessenta e apontando para eles. No nível quarto, os últimos itens do vetor V estão posicionados. O item de valor setenta do nível superior aponta setas para o de quarenta e dois e trinta e nove. O item oitenta aparece no quarto nível, mas sem nenhuma seta apontando para ele e destacada com um quadrado em volta. Uma seta tracejada sai do item treze do primeiro nível e aponta para o item quarenta e dois do quarto e último nível. Fim da descrição.

O método *heapifica*, quando invocado, vai comparando um nó pai aos respectivos nós filhos. Caso algum dos filhos seja maior que o nó pai, então, realizamos a troca entre o maior filho e o pai, de forma que, após essa operação, o nó pai seja, de fato, maior ou igual aos nós filhos para manter a propriedade da heap. Todavia, essa troca pode fazer com que o novo nó filho quebre as propriedades de heap, isto é, o nó filho, recém-trocado, pode ter novos filhos que não se categorizam enquanto heap. Por isso, é preciso invocar o método *heapifica* recursivamente, até que todos os nós necessários sejam corrigidos. Observe o passo a passo executado em método *heapifica*, na Figura 15.

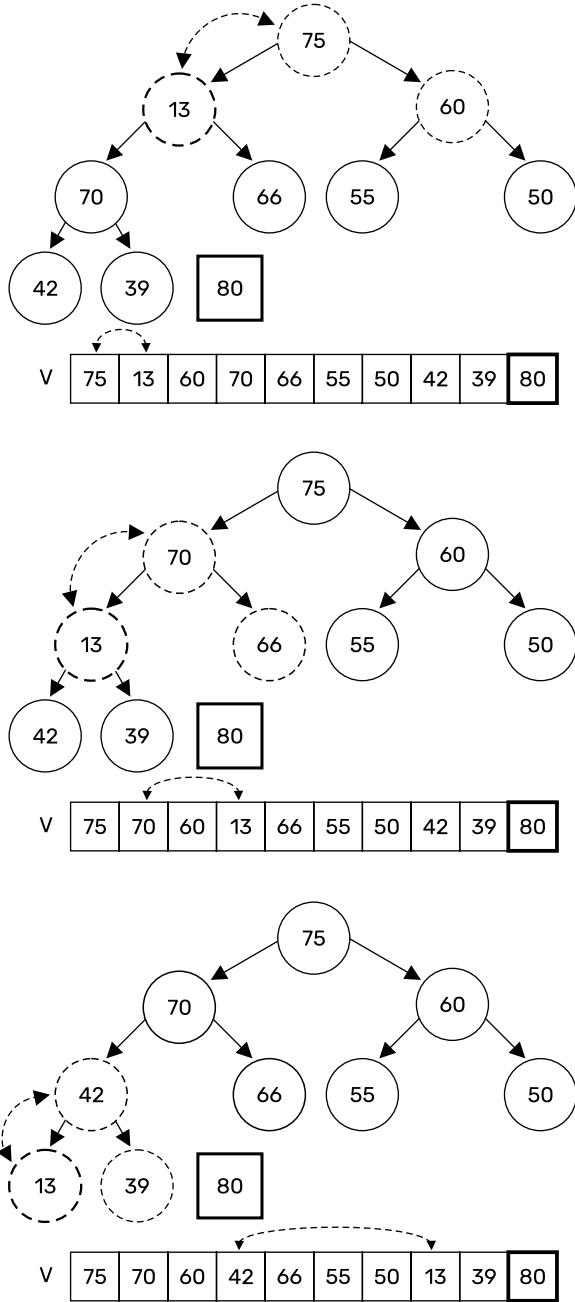


Figura 15 - Passo a passo do método *heapifica*
Fonte: os autores.

Descrição da Imagem: a figura está dividida em três figuras, cada uma representando um passo do método *heapifica*. Na primeira figura, na parte inferior da figura uma tabela com uma linha e dez colunas descreve um vetor V com os valores setenta e cinco, treze, sessenta, setenta, sessenta e seis, cinquenta e cinco, cinquenta, quarenta e dois, trinta e nove e oitenta, com um destaque para o valor oitenta. Nesse vetor, uma linha tracejada com seta nas duas pontas sai do número setenta e cinco e vai para o número 13 ao lado. Na parte superior, os itens deste vetor estão distribuídos em quatro níveis, sendo o primeiro nível apenas com o primeiro item do vetor de valor setenta e cinco envolto em um círculo tracejado. Duas setas saem do número 75, apontando para os números do nível abaixo. No segundo nível, temos o valor treze envolto em um círculo pontilhado em negrito e o sessenta envolto em um círculo pontilhado, com uma linha com setas nas duas pontas saindo do treze e indo para o setenta e cinco. No terceiro nível, estão quatro itens. Dois de valores setenta e sessenta e seis, com setas partindo do item de valor treze do nível acima até eles, e dois de valores cinquenta e cinco e cinquenta, com setas partindo do sessenta e apontando para eles. No nível quatro, os últimos itens do vetor V estão posicionados. O item de valor setenta do nível superior aponta setas para o de quarenta e dois e trinta e nove. O item oitenta aparece no quarto nível, mas sem nenhuma seta apontando para ele e destacada com um quadrado em volta. Na segunda figura, na parte inferior da figura uma tabela com uma linha e dez colunas descreve um vetor V com os valores setenta e cinco, setenta, sessenta, treze, sessenta e seis, cinquenta e cinco, cinquenta, quarenta e dois, trinta e nove e oitenta, com um destaque para o valor oitenta. Nesse vetor, uma linha tracejada com seta nas duas pontas sai do número setenta e vai para o número 13. Na parte superior, os itens deste vetor estão distribuídos em quatro níveis, sendo o primeiro nível apenas com o primeiro item do vetor de valor setenta e cinco. Duas setas saem do número 75, apontando para os números do nível abaixo. No segundo nível, temos o valor setenta envolto em um círculo pontilhado e o sessenta. No terceiro nível, estão quatro itens. Dois de valores treze e sessenta e seis envoltos em círculos pontilhados, com setas partindo do item de valor setenta do nível acima até eles e dois de valores cinquenta e cinco e cinquenta, com setas partindo do sessenta e apontando para eles. Existe uma linha com setas nas duas pontas saindo do treze e indo para o setenta. No nível quatro, os últimos itens do vetor V estão posicionados. O item de valor treze do nível superior aponta setas para o de quarenta e dois e trinta e nove. O item oitenta aparece no quarto nível, mas sem nenhuma seta apontando para ele e destacada com um quadrado em volta. Na terceira figura, na parte inferior da figura uma tabela com uma linha e dez colunas descreve um vetor V com os valores setenta e cinco, setenta, sessenta, quarenta e dois, sessenta e seis, cinquenta e cinco, cinquenta, treze, trinta e nove e oitenta, com um destaque para o valor oitenta. Nesse vetor, uma linha tracejada com seta nas duas pontas sai do número quarenta e dois e vai para o número 13. Na parte superior, os itens deste vetor estão distribuídos em quatro níveis, sendo o primeiro nível apenas com o primeiro item do vetor de valor setenta e cinco. Duas setas saem do número 75, apontando para os números do nível abaixo. No segundo nível, temos o valor setenta e o sessenta. No terceiro nível, estão quatro itens. Dois de valores quarenta e dois e sessenta e seis, sendo que o quarenta e dois está envolto em um círculo pontilhado, com setas partindo do item de valor setenta do nível acima até eles e dois de valores cinquenta e cinco e cinquenta, com setas partindo do sessenta e apontando para eles. No nível quatro, os últimos itens do vetor V estão posicionados. O item de valor quarenta e dois do nível superior aponta setas para o de treze e trinta e nove que estão envoltos em círculos tracejados. Existe uma linha com setas nas duas pontas saindo do treze e indo para o quarenta e dois. O item oitenta aparece no quarto nível, mas sem nenhuma seta apontando para ele e destacada com um quadrado em volta. Fim da descrição.

Na Figura 15, podemos notar como o método *heapifica* troca a raiz de uma subárvore com seu maior filho, à medida que é executado. Para cada nó trocado, invoca-se o método recursivamente, até que a propriedade de heap seja garantida a todos os nós envolvidos no processo. Além disso, podemos observar que, ao final, temos novamente uma heap na qual o maior entre todos os elementos se encontra na raiz.

Agora, o nó, cuja chave é igual a 75, encontra-se na raiz. Assim, repetimos o processo realizado anteriormente, no qual a raiz da heap era trocada com o último elemento do vetor – por último, entenda a última posição da porção desordenada do vetor, ou seja, a última posição da heap. Nesse caso, vamos trocar o 75 com o 13, fazendo com que o 75 se encaixe em sua posição ordenada final, de acordo com o que podemos visualizar na Figura 16.

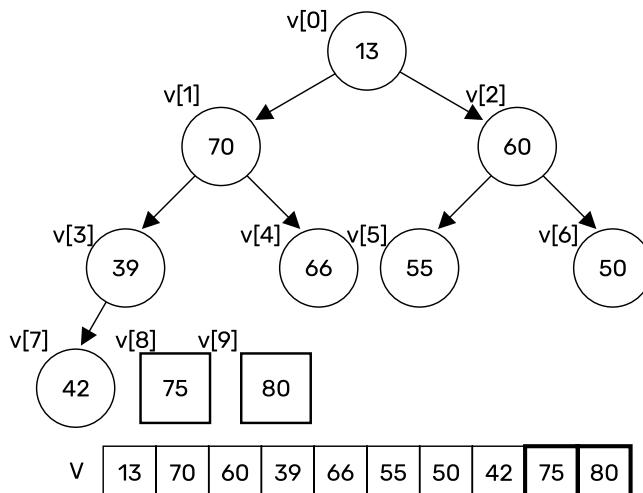


Figura 16 - Trocando o 75 com o 13 / Fonte: os autores.

Descrição da Imagem: na parte inferior da figura, uma tabela com uma linha e dez colunas descreve um vetor V com os valores treze, setenta, sessenta, trinta e nove, sessenta e seis, sessenta e cinco, sessenta e cinco, quarenta e dois, setenta e cinco e oitenta, com um destaque para o valor setenta e cinco e oitenta. Na parte superior, os itens deste vetor estão distribuídos em quatro níveis, sendo o primeiro nível apenas com o primeiro item do vetor de valor treze. No segundo nível, os valores setenta e sessenta, com setas apontando do oitenta para os dois itens inferiores. No terceiro nível, estão quatro itens. Dois de valores trinta e nove e sessenta e seis, com setas partindo do item de valor setenta do nível acima até eles, e dois de valores sessenta e cinco e sessenta e cinco, com setas partindo do sessenta e apontando para eles. No nível quatro, os últimos itens do vetor V estão posicionados. O item de valor trinta e nove do nível superior aponta uma seta para o de quarenta e dois. Os itens sessenta e cinco e oitenta aparecem no quarto nível, mas sem nenhuma seta apontando para eles e destacada com um quadrado em volta. Fim da descrição.

A partir daqui, podemos perceber que, repetindo todo o processo descrito até aqui, o Heapsort posiciona os maiores elementos ao final do arranjo, de maneira ordenada. Na Figura 16, percebemos que o valor 75 é o segundo maior entre todos os elementos de v e, corretamente, está alocado à penúltima posição do vetor.

Vejamos agora como o algoritmo Heapsort pode ser implementado em linguagem C.

```
//Garante as propriedades de heap a um nó
int heapifica(int vec[], int tam, int i){
    int e, d, maior, qtd;
    qtd = 1;
    e = 2*i+1;
    d = 2*i+2;
    if(e<tam && vec[e] > vec[i]){
        maior = e;
    }
    else {
        maior = i;
    }
    if(d<tam && vec[d] > vec[maior]){
        maior = d;
    }
    if(maior != i){
        troca(&vec[i], &vec[maior]);
        qtd += heapifica(vec, tam, maior);
    }
    return qtd;
}

//Transforma o vetor em uma heap
int constroiHeap(int vec[], int tam){
    int i, qtd;
    qtd = 0;
    for(i=tam/2;i>=0;i--){
        qtd += heapifica(vec, tam, i);
    }
    return qtd;
}
```

```
//Ordena com base na estrutura heap
int heapSort(int vec[], int tam){
    int n, i, qtd;
    qtd = 0;
    qtd += constroiHeap(vec, tam);
    n = tam;
    for(i=tam-1;i>0;i--) {
        troca(&vec[0], &vec[i]);
        n--;
        qtd += heapifica(vec, n, 0);
    }
    return qtd;
}
```



EU INDICO

Que tal aprender de uma forma divertida o funcionamento dos algoritmos vistos neste tema? Veja a seguir uma apresentação de dança simulando o funcionamento do algoritmo de ordenação Quicksort! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**



EM FOCO

Assista às aulas para complementar o conteúdo do tema.**Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

NOVOS DESAFIOS

Os algoritmos de ordenação Mergesort, Quicksort e Heapsort ainda são amplamente utilizados atualmente em diversas aplicações. Cada algoritmo tem suas próprias características e vantagens, o que os torna adequados para diferentes cenários. Vamos revisar o funcionamento de cada um e ver sua aplicabilidade?

O Mergesort é um algoritmo estável e eficiente para ordenar grandes conjuntos de dados. Ele divide a lista em duas partes, ordena cada parte separadamente e, em seguida, mescla as partes ordenadas para obter a lista final ordenada. O Mergesort é útil quando a estabilidade da ordem é importante ou quando se lida com dados distribuídos em sistemas distribuídos, onde a etapa de mesclagem pode ser paralelizada.

O Quicksort é conhecido por sua eficiência em média e é amplamente utilizado na prática. Ele escolhe um elemento como pivô e partitiona a lista em torno do pivô de forma que os elementos menores fiquem à esquerda e os maiores à direita. O Quicksort é especialmente útil para ordenação *in-place*, onde o espaço de memória é limitado, e para conjuntos de dados não uniformemente distribuídos. Em muitos casos, implementações otimizadas do Quicksort (como o Quicksort de 3 vias) são usadas para lidar com valores repetidos e melhorar o desempenho.

O Heapsort é utilizado principalmente em situações onde é necessário garantir uma complexidade de tempo constante para a ordenação. Ele constrói uma estrutura de dados chamada heap (uma árvore binária especial) e, em seguida, extrai repetidamente o elemento máximo (ou mínimo) do heap para construir a lista ordenada. O Heapsort é frequentemente usado em bibliotecas e sistemas onde a ordenação precisa ser previsível e rápida, independentemente dos dados de entrada.

Com o avanço da tecnologia e das bibliotecas de programação, muitas bibliotecas e frameworks modernos oferecem funções de ordenação otimizadas e testadas, que escolhem automaticamente o algoritmo mais adequado para os dados de entrada e o contexto de uso. Ainda assim, é importante conhecer como esses algoritmos são implementados para conhecer o funcionamento interno dessas funções.

VAMOS PRATICAR

- O algoritmo Quicksort é um método de ordenação eficiente e amplamente utilizado que se destaca por sua velocidade e desempenho em muitos cenários e é uma das opções preferidas para ordenar grandes conjuntos de dados. O Quicksort se baseia na estratégia de “divisão e conquista” para realizar a ordenação. Nesse algoritmo, um elemento da lista é selecionado como pivô. Os elementos da lista são, então, rearranjados de forma que os elementos menores que o pivô fiquem à esquerda e os elementos maiores, à direita. Isso é feito através de uma operação de particionamento. O Quicksort é aplicado recursivamente a cada uma das partições resultantes, ordenando-as separadamente.

O que uma árvore precisa para ser considerada uma árvore estritamente binária?

- A técnica de ordenação Mergesort utiliza um conceito conhecido por dividir para conquistar. Esse conceito sugere que um problema complexo possa ser dividido em dois problemas menores, e cada um desses seja dividido novamente em partes menores ainda, até que se encontre uma parte pequena e simples suficiente para que seja resolvido.

Explique como o algoritmo Mergesort aplica a técnica dividir para conquistar.

- O algoritmo de ordenação Mergesort é uma abordagem eficiente e elegante para organizar elementos em uma lista, utilizando uma técnica de divisão e conquista. A abordagem adotada pelo Mergesort torna-o altamente eficiente em termos de tempo de execução, especialmente para listas de tamanho considerável.

Indique o funcionamento do Mergesort:

- O Mergesort divide a lista em subarrays de tamanho igual, ordena cada subarray individualmente e, em seguida, mescla os subarrays ordenados em uma única lista ordenada.
- O Mergesort seleciona o elemento mínimo em cada iteração e o coloca na posição apropriada na lista.
- O Mergesort compara elementos adjacentes e troca elementos fora de ordem até que a lista esteja completamente ordenada.

VAMOS PRATICAR

- d) O Mergesort move os elementos um por um para a parte ordenada da lista, construindo gradualmente a lista ordenada
 - e) O Mergesort usa um conjunto de incrementos variáveis para otimizar a ordenação dos elementos na lista.
4. A estrutura de dados Heap é uma das ferramentas fundamentais na área de Ciência da Computação e é amplamente utilizada para lidar com prioridades e ordenação eficiente de dados. Ela é uma árvore especializada que segue uma propriedade intrínseca, conhecida como propriedade de Heap, que garante que o elemento de maior (ou menor) valor esteja sempre localizado na raiz da árvore. Essa característica é extremamente útil em algoritmos de ordenação.

Qual é a relação entre a estrutura de dados Heap e o algoritmo Heapsort?

- a) A estrutura de dados Heap é usada pelo algoritmo Heapsort para armazenar elementos ordenados.
 - b) A estrutura de dados Heap é uma variação do algoritmo Heapsort, projetada para melhor desempenho em listas grandes.
 - c) O algoritmo Heapsort utiliza a estrutura de dados Heap para manter uma propriedade específica, que é fundamental para o seu funcionamento eficiente.
 - d) A estrutura de dados Heap é uma alternativa menos eficaz ao algoritmo Heapsort, geralmente preferida apenas para fins didáticos.
 - e) A estrutura de dados Heap é uma parte opcional do algoritmo Heapsort e não tem impacto significativo no seu desempenho.
5. Os algoritmos Quicksort, Mergesort e Heapsort são três métodos de ordenação amplamente utilizados, cada um com suas próprias estratégias distintas. O Quicksort se destaca por sua eficiência média, enquanto o Mergesort garante uma ordenação estável e previsível. Por outro lado, o Heapsort proporciona um desempenho confiável em operações de prioridade. Cada algoritmo tem suas características únicas que os tornam adequados para diferentes cenários de aplicação, permitindo aos desenvolvedores escolher a abordagem mais adequada às necessidades específicas.

Analise as afirmativas a seguir sobre as semelhanças entre os algoritmos de ordenação Quicksort, Mergesort e Heapsort:

VAMOS PRATICAR

- I - Todos os três métodos utilizam a estratégia de “divisão e conquista” em seus processos.
- II - Heapsort utiliza a estrutura de dados Heap para facilitar a ordenação.
- III - Os três métodos podem ser implementados de forma recursiva.
- IV - Todos os três algoritmos são baseados em comparações entre elementos.

É correto o que se afirma em:

- a) I e IV, apenas.
- b) II e III, apenas.
- c) III e IV, apenas.
- d) I, II e III, apenas.
- e) II, III e IV, apenas.

REFERÊNCIAS

CORMEN, T. H. *et al.* **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002.

GABARITO

1. A escolha do pivô é um dos aspectos fundamentais e sensíveis no algoritmo de ordenação Quicksort. A eficácia da escolha do pivô tem um impacto direto no desempenho do algoritmo, influenciando o número de comparações e trocas realizadas durante o processo de ordenação.

O pivô é o elemento usado para dividir a lista em duas partições durante o processo de particionamento. A ideia central é posicionar o pivô de forma que os elementos menores que ele fiquem à sua esquerda e os elementos maiores, à sua direita. O objetivo é criar partições bem equilibradas para que o algoritmo funcione de maneira eficiente.

No entanto, a escolha do pivô pode ser um desafio, pois uma escolha inadequada pode levar a partições desequilibradas, aumentando o número de comparações e prejudicando a eficiência do Quicksort.

2. O algoritmo Mergesort faz isso de forma recursiva. Assim que o vetor é dividido, cada uma das metades é passada como parâmetro a uma nova chamada da função Mergesort. Essa recursividade desce até o ponto em que o vetor tem apenas um único valor. Nesse momento, inicia-se o retorno da recursividade, e os vetores unitários são comparados e unidos aos já ordenados.

3. **Opção A.** O algoritmo usa a técnica de dividir para conquistar. Ele pega uma lista muito grande e divide-o em duas listas menores. Aplica o resultado novamente no algoritmo, dividindo-o em partes cada vez menores. Quando não houver mais possibilidade de divisão, a técnica começa a remontar a lista, dessa vez já ordenada.

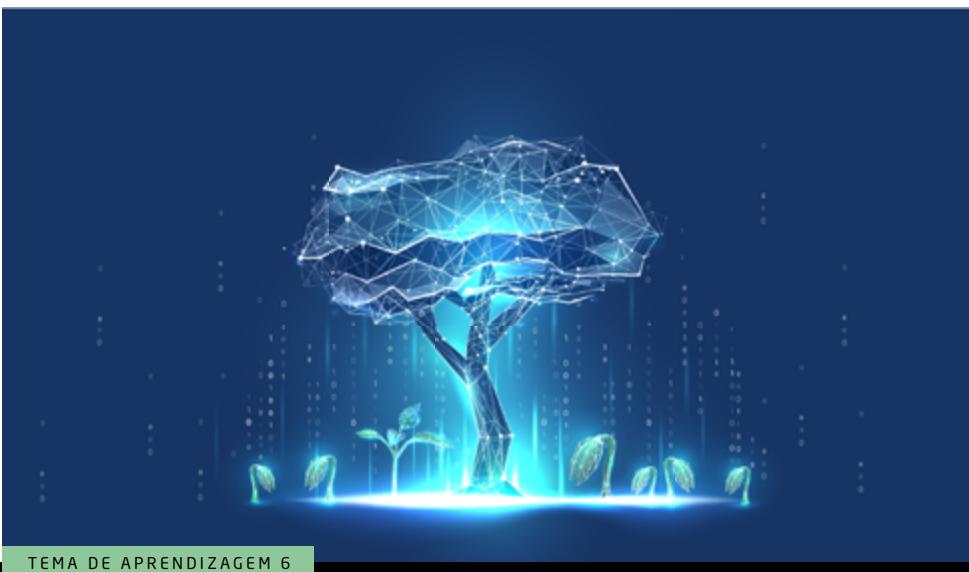
4. **Opção C.** A heap é uma árvore binária completa (ou quase completa) que sempre armazena em sua raiz o elemento de maior valor entre todos os dados armazenados. Assim, no Heapsort, podemos nos aproveitar dessa propriedade para realizar a ordenação dos dados de modo a trocar a raiz com a última posição não ordenada do vetor.

5. **Opção E.** A técnica de ordenação Mergesort utiliza um conceito conhecido por dividir para conquistar. Esse conceito sugere que um problema complexo possa ser dividido em dois problemas menores, e cada um desses sejam divididos novamente em partes menores ainda, até que se encontre uma parte pequena e simples suficiente para que seja resolvido. O algoritmo Mergesort faz isso de forma recursiva. Assim que o vetor é dividido, cada uma das metades é passada como parâmetro a uma nova chamada da função Mergesort.

GABARITO

Essa recursividade desce até o ponto em que o vetor tem apenas um único valor. Nesse momento, inicia-se o retorno da recursividade, e os vetores unitários são comparados e unidos já ordenados.

O Quicksort também utiliza a estratégia de dividir para conquistar. O primeiro passo é escolher um elemento qualquer que será denominado de pivô. A partir desse elemento, a lista será dividida em três sublistas, uma para o pivô, uma para os valores menores e outra para os valores maiores do que o próprio pivô. Isso garante que as chaves menores precedam as chaves maiores e que o pivô esteja na sua correta posição dentro do vetor. O algoritmo Heapsort se aproveita da característica do heap para realizar a ordenação em um vetor.



TEMA DE APRENDIZAGEM 6

ÁRVORES

MINHAS METAS

- Conhecer a estrutura de árvore binária.
- Aprender sobre árvore estritamente binária.
- Reconhecer uma árvore binária completa.
- Implementar uma árvore binária em linguagem C.
- Conhecer uma forma diferente de estruturar uma árvore binária.

INICIE SUA JORNADA

Neste tema de aprendizagem, estudaremos uma estrutura de dados útil para várias aplicações: a árvore binária. Definiremos algumas formas dessa estrutura de dados e mostraremos como podem ser implementadas na linguagem C.

A árvore como estrutura é muito utilizada para organizar informações armazenadas tanto na memória principal como na secundária. Isso se dá devido ao fato de ser fácil e rápida a pesquisa de dados em árvores.

Imagine um sistema de gerenciamento de uma livraria on-line, onde os clientes podem comprar e alugar livros digitais. O sistema deseja oferecer recomendações personalizadas aos usuários com base em seus interesses de leitura. O desafio é fazer recomendações de maneira eficiente, de modo que os livros recomendados estejam diretamente relacionados aos interesses do usuário e permitam uma busca rápida e eficaz por recomendações relevantes.

A livraria pode implementar essa solução usando uma árvore binária. Os nós da árvore representarão livros, com os nós filhos esquerdo e direito representando as recomendações. Ao cadastrar um novo cliente e coletar suas preferências de leitura, o sistema pode inserir os livros relevantes na árvore, garantindo que os nós recomendados estejam próximos aos nós pais, facilitando a busca.

Essa implementação não apenas melhora a experiência do cliente, mas também demonstra como a estrutura de árvores binárias pode ser aplicada de forma prática para resolver problemas do mundo real. A eficiência na construção e busca na árvore binária é crucial para garantir que os usuários possam acessar facilmente recomendações que atendam aos seus interesses, resultando em um sistema de recomendação eficaz e bem-sucedido.

Neste tema, daremos foco no entendimento da estrutura e não na sua definição matemática.

PLAY NO CONHECIMENTO

As árvores binárias demonstram sua aplicabilidade em uma variedade de cenários reais, desde o gerenciamento de arquivos até a otimização de redes e sistemas de banco de dados. Quer conhecer essas aplicações em detalhes? Então, dê o play e nos acompanhe nessa jornada! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

DESENVOLVA SEU POTENCIAL

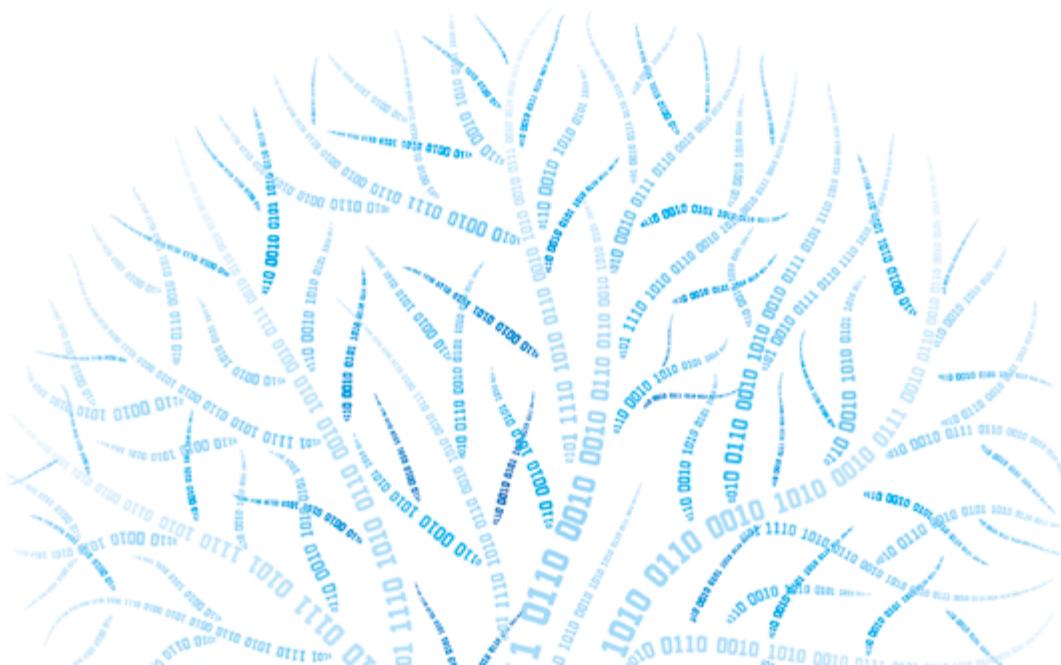
ÁRVORES BINÁRIAS

Segundo Tenenbaum (1995, p. 303):



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

A Figura 1 apresenta um exemplo de árvore binária. Essa árvore possui 9 nós, sendo A o nó raiz da árvore. Ela possui uma subárvore esquerda com raiz em B e uma direita com raiz em C. A subárvore com raiz em B possui uma subárvore esquerda com raiz em D, e a direita com raiz em E. A subárvore com raiz em C possui uma subárvore esquerda vazia e uma subárvore direita com raiz em F. As árvores binárias com raiz em D, G, H e I possuem subárvore direita e esquerda vazias.



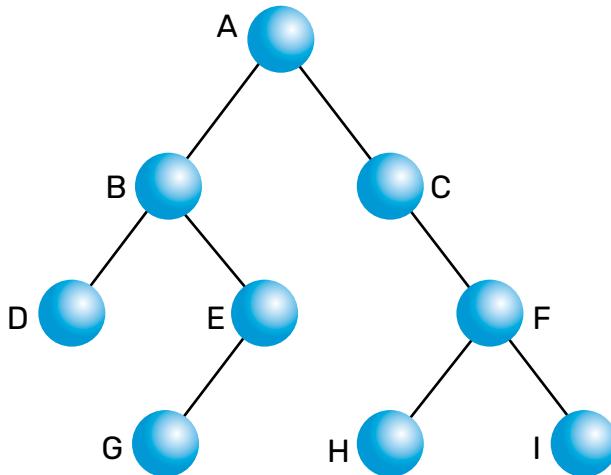


Figura 1 - Exemplo de árvore binária / Fonte: o autor.

Descrição da Imagem: a figura possui quatro níveis, um abaixo do outro. No primeiro nível, há apenas um item marcado com a letra A, ligado por uma linha a dois itens do nível dois de letras B e C. O item B está ligado, também, a dois itens do nível três de letras D e E. O item C está ligado a um item do nível três de letra F. O item E está ligado a um item do nível quatro de letra G e o item F está ligado a dois itens do quarto nível de letras H e I. A distribuição desses itens remete à forma de um triângulo, tendo mais itens na base do que no topo. Fim da descrição.

Se A é o nó raiz de uma árvore binária e tem uma subárvore com raiz em B, então, se diz que A é pai de B e B é filho esquerda de A. Na mesma forma, se A tem uma subárvore direita com raiz em C, diz-se que A é pai de C e C é filho direita de A. Um nó sem filhos é chamado de folha. Na árvore representada pela Figura 1, os nós D, G, H e I são considerados folhas.

Os nós D e E são ambos filhos de B, dessa forma, podemos considerá-los como nós irmãos. Também podemos classificar os nós de acordo com a sua hereditariedade. O nó B é ancestral dos nós D, E e G, da mesma forma que esses três nós são descendentes de B.

ZOOM NO CONHECIMENTO

Uma árvore binária é um tipo de grafo que tem regras específicas na sua construção. Cada nó tem no máximo dois filhos e um único pai, excetuando-se o nó raiz da árvore principal, que é órfão.

Quando se percorre uma árvore a partir da raiz em direção às folhas, diz-se que se está **caminhando para baixo**, ou descendo na árvore. De forma análoga, quando se está percorrendo uma árvore a partir de uma de suas folhas em direção à raiz, diz-se que se está **caminhando para cima**, ou subindo na árvore. Se fôssemos fazer uma analogia entre uma árvore na informática e uma árvore real (planta), na computação as árvores seriam representadas de cabeça para baixo.

Árvore Estritamente Binária

Uma árvore é considerada estritamente binária se todo nó que não for folha tiver sempre subárvores direita e esquerda não vazias. Na Figura 2, temos um exemplo de árvore estritamente binária. São considerados folhas os nós C, D, F e G. Os nós A, B e E possuem subárvores esquerda e direita não vazias.

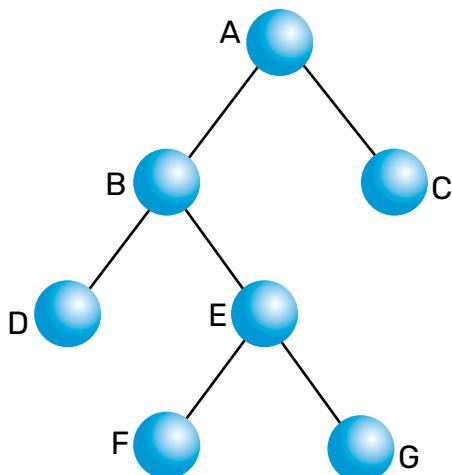


Figura 2 - Exemplo de árvore estritamente binária
Fonte: o autor.

Descrição da Imagem: a figura tem quatro níveis, um abaixo do outro. No primeiro nível, há apenas um item marcado com a letra A, ligado por uma linha a dois itens do nível dois de letras B e C. O item B está ligado também a dois itens do nível três de letras D e E. O item E está ligado a um item do nível quatro de letra G. Cada item só está ligado a no máximo dois itens do nível inferior. Fim da descrição.

A árvore apresentada na Figura 1 não pode ser considerada uma árvore estritamente binária já, posto que o nó C possui uma subárvore esquerda vazia e o nó E possui, uma subárvore direita vazia.

A quantidade de nós presentes numa árvore estritamente binária se dá pela fórmula $n = (2 * f) - 1$ em que f é o número de folhas na árvore. Vamos testar essa fórmula a partir da árvore representada pela Figura 2.

A árvore possui 4 folhas, quais sejam os nós: C, D, F e G. Aplicando-se a fórmula, temos:

$$n = (2 * f) - 1$$

Como f é igual a 4, substituímos o valor de f na fórmula:

$$n = (2 * 4) - 1$$

Realizamos a multiplicação e, em seguida, a subtração.

$$n = (8) - 1$$

Temos, então, que:

$$n = 7$$

A árvore possui exatamente 7 nós: A, B, C, D, E, F e G.

Também podemos calcular a quantidade de folhas em uma árvore estritamente binária pela dedução da fórmula:

$$n = (2 * f) - 1$$

Adicionando 1 em ambos os lados da igualdade.

$$n + 1 = (2 * f) - 1 + 1$$

Efetuando o cálculo:

$$n + 1 = (2 * f)$$

Dividindo ambos os lados da igualdade por 2:

$$\frac{n+1}{2} = \frac{(2 * f)}{2}$$

Efetuando o cálculo:

$$\frac{n+1}{2} = f$$

Vamos inverter os lados da igualdade para deixar a variável isolada à esquerda:

$$f = \frac{n+1}{2}$$

No caso da árvore estritamente binária da Figura 2, sabemos que possui 7 nós. Colocando na fórmula:

$$f = \frac{7+1}{2}$$

Realizando a soma e, em seguida, a divisão:

$$f = \frac{8}{2}$$

Temos que:

$$f = 4$$

A árvore possui exatamente 4 folhas: C, D, F e G.

Árvore Binária Completa

O nó raiz de uma árvore binária é considerado como de nível 0. A partir dele, cada nó possui um nível a mais do que o seu pai. Por exemplo, na árvore binária da Figura 3, o nó C está no nível 1, F está no nível 2 e o nó M, no nível 3. A profundidade ou altura de uma árvore binária é dada pelo maior nível de qualquer folha na árvore. Isso equivale ao tamanho do percurso mais distante da raiz até uma folha qualquer. Dessa forma, a profundidade da árvore da Figura 3 é 3.

Quando uma árvore estritamente binária possui todas as folhas no último nível, ela é chamada de árvore binária completa. A Figura 3 demonstra uma **árvore binária completa de profundidade 3**.

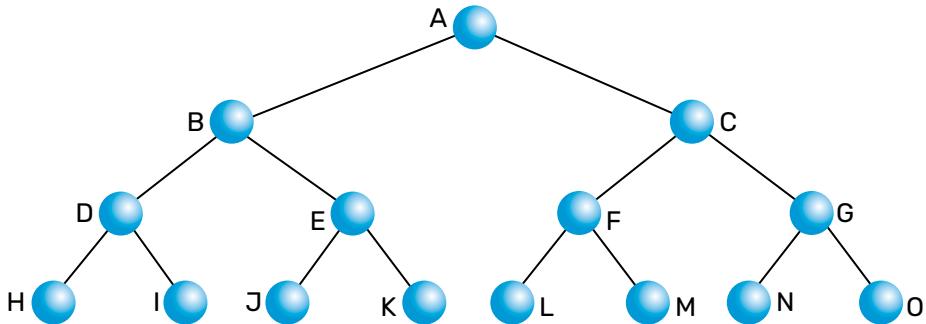


Figura 3 - Exemplo de árvore binária completa / Fonte: o autor.

Descrição da Imagem: a figura possui quatro níveis, um abaixo do outro. No primeiro nível há apenas um item marcado com a letra A, ligado por uma linha a dois itens do nível dois de letras B e C. O item B está ligado, também, a dois itens do nível três de letras D e E. O item C está ligado a dois itens do nível três de letras F e G. O item D está ligado a dois itens do nível quatro de letras H e I. O item E está ligado a dois itens do nível quatro de letras J e K. O item F está ligado a dois itens do nível quatro de letras L e M. E o item G está ligado a dois itens do nível quatro de letras N e O. A distribuição desses itens remete à forma de um triângulo, tendo mais itens na base do que no topo. Fim da descrição.

Implementando Árvore Binária em C

Existem várias formas de implementar uma árvore binária. A mais simples delas é **usar um vetor de nós**. Cada nó possui, pelo menos, três valores: pai, esquerda e direita. O atributo *pai* vai apontar para a posição no vetor do pai do nó. O atributo *esquerda* vai armazenar a posição da raiz da subárvore esquerda, e o atributo *direita* guarda a posição da raiz da subárvore direita. Vamos adicionar mais um atributo, *dado*, que irá armazenar o valor do nó.

```

//Estrutura
struct str_no {
    char dado;
    int esquerda;
    int direita;
    int pai;
};
  
```

Com a estrutura já definida, vamos criar uma variável do tipo *vetor* de *str_no*. Esse vetor terá o tamanho definido por uma constante chamada *tamanho*. Precisaremos, também, de uma variável do tipo inteiro que servirá de índice para o nosso vetor.

```
//Constantes
#define tamanho 100

//Variáveis
struct str_no arvore[tamanho];
int indice=0;
```

Para inserir um nó na árvore, é **necessário** saber o seu valor, quem é o seu pai e se ele é um filho esquerda ou direita. Mesmo sabendo quem é o pai, antes de fazer a inserção no vetor, é preciso encontrar a sua localização. Para tanto, criou-se uma função chamada *arvore_procura*, que retorna um valor inteiro (posição no vetor) e tem, como parâmetro, o nome do pai.

```
//Procura nó
int arvore_procura(char dado) {
    if (indice != 0) {
        for (int i = 0; i<indice; i++) {
            if (arvore[i].dado == dado) {
                return (i);
            }
        }
    }
    else {
        return (0);
    }
}
```

Note que este trecho de programa apresentado faz uma verificação no valor da variável *indice*. Se o valor for 0, significa que a árvore está vazia e o valor a ser inserido será a raiz da árvore. **A função leva em conta que o pai está presente na árvore, já posto que não** foi previsto nenhum tipo de retorno de erro para o caso de o pai não ser encontrado.

Já foi realizada a leitura do valor a ser inserido e já se sabe quem é o seu pai e qual a sua descendência. Por meio da função *arvore_procura*, passa-se o valor do pai como parâmetro e obtém como retorno a sua posição no vetor. Agora é possível realizar a inserção do novo nó na árvore.

Para tanto, criou-se a função chamada *arvore_insere*. Ela terá como parâmetro um valor inteiro que representa a posição do pai no vetor, o valor dado digitado pelo usuário e a sua posição de descendência (se é filho do lado esquerdo ou direito).

```
1      //Inserir nó
2      void arvore_insere(int pai, char dado, int lado){
3          switch (lado){
4              case E:
5                  arvore[pai].esquerda = indice;
6                  arvore[indice].dado = dado;
7                  arvore[indice].pai = pai;
8                  arvore[indice].esquerda = -1;
9                  arvore[indice].direita = -1;
10                 indice++;
11                 break;
12             case D:
13                 arvore[pai].direita = indice;
14                 arvore[indice].dado = dado;
15                 arvore[indice].pai = pai;
16                 arvore[indice].esquerda = -1;
17                 arvore[indice].direita = -1;
18                 indice++;
19                 break;
20         }
21     }
```

Na função *arvore_insere*, criou-se uma variável chamada indice, que guarda a primeira posição livre da arvore. O parâmetro pai recebido na função indica qual a posição do nó pai. Se o novo nó for filho esquerdo, será atribuído o valor de indice ao atributo esquerdo na arvore, na posição pai (linha 5).

No caso de filho direito, colocou-se o valor de *indice* no atributo *direito* (linha 13). O próximo passo é guardar o nome do nó no atributo *dado* e a referência do pai. Para tanto, marca-se com -1 os valores de esquerda e direita para identificar que ambos os ponteiros não apontam para uma subárvore. Esses passos estão demonstrados nas linhas 6 a 10 para filho esquerda e 14 a 18 para filho direita.

O programa, a seguir, traz uma implementação completa de uma árvore binária em linguagem C. Ele traz a declaração de bibliotecas, constantes e variáveis auxiliares, função principal, função para desenhar o menu de opções, entre muitas outras linhas de código.

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

//Constantes
#define tamanho 100
#define E 0
#define D 1
#define R -1

//Estrutura
struct str_no {
    char dado;
    int esquerda;
    int direita;
    int pai;
};

//Variáveis
struct str_no arvore[tamanho];
int lado, indice=0;
int opt=-1;
char pai, no;
```

```
//Prototipação
void arvore_insere(int pai, char dado, int lado);
int arvore_procura(char dado);
void menu_mostrar(void);

//Função principal
int main(void){
    int temp;
    do {
        menu_mostrar();
        scanf("%d", &opt);
        switch (opt) {

            case 1:
                printf("\nDigite o valor do PAI: ");
                scanf("%c", &pai);
                scanf("%c", &pai);
                printf("Digite o valor do NO: ");
                scanf("%c", &no);
                scanf("%c", &no);
                printf("Digite o lado da subarvore (E=%d/D=%d/
R=%d): ",
E,D,R);
                scanf("%d", &lado);
                temp = arvore_procura(pai);
                arvore_insere(temp,no,lado);
                break;

            case 2:
                printf("Digite o valor do NO: ");
                scanf("%c", &no);
                scanf("%c", &no);
                temp = arvore_procura(no);
                printf("No %c\nFilho Esquerda: %c
\nFilho Direita: %c\n\n",
arvore[temp].dado,
arvore[arvore[temp].esquerda].dado,
arvore[arvore[temp].direita].dado);
                system("pause");
                break;
        }
    }
}
```

```
    }while (opt!=0);
    system("pause");
    return(0);
}

//Inserir nó
void arvore_insere(int pai, char dado, int lado){
    switch (lado){

        case E:
            arvore[pai].esquerda = indice;
            arvore[indice].dado = dado;
            arvore[indice].pai = pai;
            arvore[indice].esquerda = -1;
            arvore[indice].direita = -1;
            indice++;
            break;
        case D:
            arvore[pai].direita = indice;
            arvore[indice].dado = dado;
            arvore[indice].pai = pai;
            arvore[indice].esquerda = -1;
            arvore[indice].direita = -1;
            indice++;
            break;
        case R:
            arvore[indice].dado = dado;
            arvore[indice].pai = -1;
            arvore[indice].esquerda = -1;
            arvore[indice].direita = -1;
            indice++;
            break;
    }
}
```

```
//Procura nó
int arvore_procura(char dado) {
    if (indice != 0) {
        for (int i = 0; i < indice; i++) {
            if (arvore[i].dado == dado) {
                return (i);
            }
        }
    }
    else {
        return (0);
    }
}

//Desenha o menu na tela
void menu_mostrar(void) {
    system("cls");
    for (int i = 0; i < indice; i++) {
        printf("| %c ", arvore[i].dado);
    }
    printf("\n1 - Inserir um NO na arvore");
    printf("\n2 - Pesquisar um NO na arvore");
    printf("\n0 - Sair...\n\n");
}
```

Vimos uma forma de armazenar uma árvore binária em um vetor. Essa implementação é bem simples e rápida, mas longe do ideal. Voltamos ao mesmo problema de estruturas anteriores em que muita memória é alocada na execução do programa para uma aplicação que pode ou não precisar de todo o espaço reservado na memória.

O programa anteriormente apresentado procura sempre a primeira posição livre no vetor para posicionar um novo nó na árvore. Dessa forma, a ordenação da estrutura estará diretamente ligada à ordem que os nós foram adicionados.

Uma Árvore Binária Diferente

Outra forma de armazenar uma árvore binária em um vetor é reservar as posições de acordo com o nível e descendência de cada nó. O primeiro nó a ser armazenado é a raiz da árvore e ele ficará na primeira posição do vetor, lembrando que os vetores em C começam na posição 0.

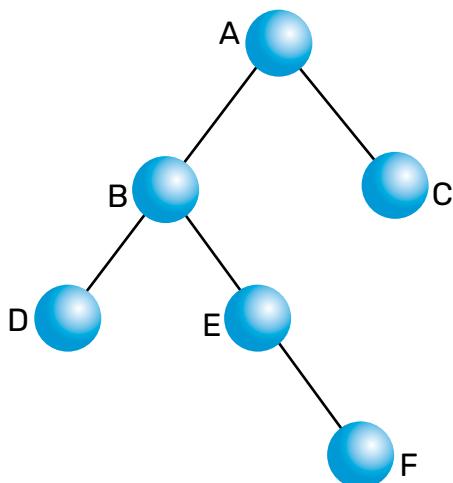


Figura 4 - Árvore binária com 6 nós e profundidade 3
Fonte: o autor.

Descrição da Imagem: a figura possui três níveis, um abaixo do outro. No primeiro nível, há apenas um item marcado com a letra A, ligado por uma linha a dois itens do nível dois de letras B e C. O item B está ligado também a dois itens do nível três de letras D e E. O item E está ligado a um item do nível quatro de letra F. Fim da descrição.

Vamos simular a inserção da árvore representada pela Figura 4 num vetor de 16 posições. O primeiro nó a ser inserido será a raiz da árvore A e ocupará a posição 0 do vetor:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A															

Como nosso objetivo é manter a árvore indexada dentro do vetor, vamos reservar a primeira posição **p** logo após o nó para armazenar o filho esquerdo e a segunda posição **p** para o filho direito. Assim, para um nó armazenado numa posição **p** qualquer, seu filho esquerdo estará na posição **p+1** e seu filho direito na posição **p+2**. Vamos inserir os nós B e C no vetor.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C													

O nó A está na posição $p=0$, então é esperado que seu filho esquerdo esteja na posição $p+1=1$ e o filho direito na posição $p+2=2$ do vetor, o que é verdade. Vamos inserir agora os nós D e E.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E											

Usando a fórmula que foi proposta, para encontrar os filhos de B na posição 1 do vetor, o filho esquerdo estará na posição $p+1=2$ e o direito na posição $p+2=3$, o que não é verdade, logo a fórmula que propusemos não serve para resolver o problema de ordenação de árvore binária em vetor de dados.

Uma árvore binária cresce de forma geométrica posto que cada nó tem dois filhos, que, por sua vez, são duas subárvore que podem estar vazias ou não. Independentemente de o filho existir, seu espaço precisa ser reservado no vetor.

Como todo nó em uma árvore binária tem dois filhos, vamos modificar a fórmula para $2*p+1$, para o filho esquerdo, e $2*p+2$, para o filho direito, sendo p a posição do nó no vetor.

Aplicando as novas fórmulas para o nó A que está na posição 0, temos que $2*p+1=1$ é a posição de B (filho esquerdo de A) e $2*p+2=2$ é a posição de C (filho direito de A).

Para B, que está na posição 1, temos que $2*p+1=3$, que é a posição de D (filho esquerdo de B) e $2*p+2=4$ é a posição de E (filho direito de B). O nó C é uma folha, podemos dizer que ele não tem filhos ou que seus filhos são árvores vazias. Como o nosso objetivo é manter o vetor ordenado, a posição dos filhos de C será reservada aplicando a fórmula proposta. Assim, a posição $2*p+1=5$ ficará disponível para o filho esquerdo de C e $2*p+2=6$, para o seu filho direito.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-									

O próximo será D, que também é uma folha. Então, será reservado no vetor as posições $2*p+1=7$ e $2*p+2=8$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-	-	-							

O último nó de nível 2 (E) possui um filho direito (F), que será armazenado na posição $2*p+2=10$ do vetor criado para a nossa árvore binária.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	-	-	-	-	-	F					

No primeiro momento, podemos pensar que essa não é uma boa abordagem. Se a árvore não for binária completa, existirão vários espaços vazios no vetor, lembrando uma memória fragmentada. Por outro lado, ocuparemos exatamente a mesma quantidade de memória que a implementação anterior. A principal diferença é que, nesse modelo, os nós da árvore estarão indexados, assim, é possível obter as informações de forma rápida e precisa, utilizando-se do índice, em vez de percorrer toda a estrutura.

ZOOM NO CONHECIMENTO

Uma árvore binária também pode ser criada dinamicamente. Basta ter um ponteiro esquerdo e direito que apontará para os filhos e um ponteiro para o pai.

 EM FOCO

Assista às aulas para complementar seus conhecimentos sobre este tema.

Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.

NOVOS DESAFIOS

Neste tema, foi apresentada uma nova estrutura de dados: a árvore binária. Ela lembra muito um grafo, porém essas duas estruturas se diferem pelos seguintes motivos:

3. O grafo pode não ter nenhuma aresta e contém no mínimo um único vértice.
4. Uma árvore pode ser vazia e todos os nós podem ser de no máximo grau 3, sendo que cada nó tem um único pai e dois filhos.

Vimos duas formas de representar essa poderosa estrutura pelo uso de vetores. No primeiro modelo, o vetor vai sendo preenchido à medida que a árvore é lida. Cada nó entra na árvore na primeira posição livre da variável. A segunda implementação possui uma abordagem focada em indexar os nós dentro do vetor, posicionando-os de acordo com a sua ascendência e descendência.

As árvores binárias têm uma importância significativa nas implementações de programas devido às suas propriedades e versatilidade. Elas são amplamente utilizadas em várias aplicações práticas, como sistemas de gerenciamento de bancos de dados, para criar índices que aceleram a busca e a recuperação de informações em grandes conjuntos de dados; para a criação de árvores de decisão em algoritmos de aprendizado de máquina para classificar dados com base em um conjunto de regras definidas nos nós; na renderização gráfica em computação gráfica para estruturar a renderização de cenas em gráficos 3D.

VAMOS PRATICAR

- As árvores binárias são uma estrutura de dados fundamental em Ciência da Computação, amplamente utilizadas para representar hierarquias e organizar informações de maneira eficiente. Uma árvore binária é uma coleção de nós conectados, onde cada nó possui no máximo dois filhos: um filho esquerdo e um filho direito. Essa estrutura permite representar uma ampla variedade de problemas, desde estruturas de dados até algoritmos de pesquisa e ordenação (CORMEN, 2002).

Fonte: CORMEN, T. H. *et al. Algoritmos: teoria e prática.* Rio de Janeiro: Campus, 2002. v. 2.

O que uma árvore precisa para ser considerada uma árvore estritamente binária?

- O nó raiz de uma árvore binária é considerado como de nível 0. A partir dele, cada nó possui um nível a mais do que o seu pai. A profundidade ou altura de uma árvore binária é dada pelo maior nível de qualquer folha na árvore. Isso equivale ao tamanho do percurso mais distante da raiz até uma folha qualquer.

Quais são as características de uma árvore binária completa?

- “Uma árvore binária é um conjunto finito de elementos que está vazio ou é partitionado em três subconjuntos disjuntos. O primeiro subconjunto contém um único elemento, chamado raiz da árvore. Os outros dois subconjuntos são em si mesmos árvores binárias, chamadas subárvores esquerda e direita da árvore original. Uma subárvore esquerda ou direita pode estar vazia. Cada elemento de uma árvore binária é chamado nó da árvore” (TENENBAUM, 1995, p. 303).

O que caracteriza uma árvore binária?

- a) Cada nó tem exatamente um filho.
- b) Cada nó tem, no máximo, um filho.
- c) Cada nó tem exatamente dois filhos.
- d) Cada nó tem, no máximo, dois filhos.
- e) Cada nó tem três filhos.

VAMOS PRATICAR

4. A profundidade ou altura de uma árvore binária é dada pelo maior nível de qualquer folha na árvore. Isso equivale ao tamanho do percurso mais distante da raiz até uma folha qualquer.

Considere uma árvore binária com altura 3, onde o nível 0 é o nível da raiz. Quantos nós no total podem ter essa árvore?

- a) 3 nós.
- b) 7 nós.
- c) 15 nós.
- d) 31 nós.
- e) 63 nós.

5. Uma árvore é considerada estritamente binária se todo nó que não for folha tiver sempre subárvore direita e esquerda não vazias (CORMEN, 2002).

Fonte: CORMEN, T. H. et al. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. v. 2.

Com base nas informações apresentadas, avalie as asserções a seguir e a relação proposta entre elas:

- I - Uma árvore estritamente binária é uma árvore binária em que cada nó interno tem exatamente dois filhos.

PORQUE

- II - As árvores completas são estritamente binárias.

A respeito dessas asserções, assinale a opção correta:

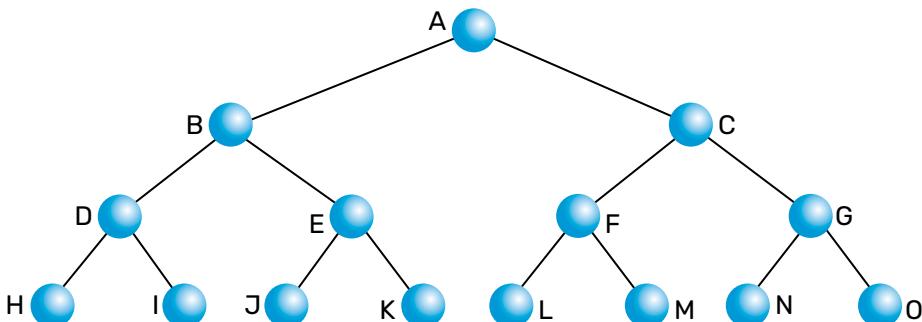
- a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
- b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
- c) A asserção I é uma proposição verdadeira e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa e a II é uma proposição verdadeira.
- e) As asserções I e II são falsas.

REFERÊNCIAS

TENENBAUM, A. M. *et al.* **Estruturas de dados usando C**. São Paulo: Makron Books, 1995.

GABARITO

1. Uma árvore é considerada estritamente binária se todo nó que não for folha tiver sempre subárvore direita e esquerda não vazias.
2. Uma árvore é dita binária completa quando for uma árvore estritamente binária e todas as suas folhas estiverem no último nível da estrutura.
3. **Opcão D.** Uma árvore binária é um tipo de grafo que tem regras específicas na sua construção. Cada nó tem no máximo dois filhos e um único pai, excetuando-se o nó raiz da árvore principal, que é órfão.
4. **Opcão C.** Considerando uma árvore binária com altura 3, sendo o nível 0 a raiz, essa árvore pode ter até 15 nós, conforme a seguinte figura abaixo:

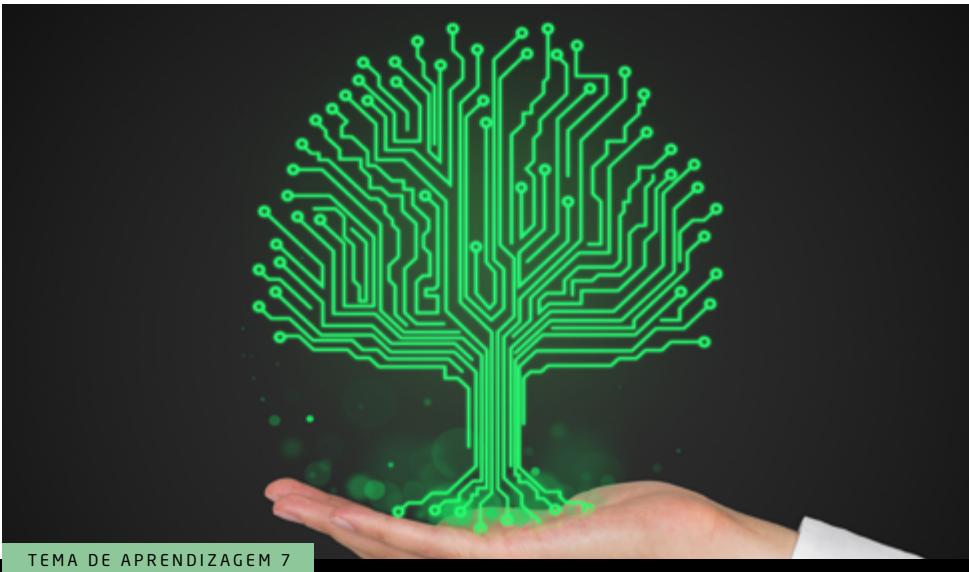


5. **Opcão A.** Uma árvore estritamente binária é aquela em que cada nó interno tem exatamente dois filhos. Portanto, em uma árvore completa, onde todos os nós, exceto possivelmente os do último nível, têm dois filhos, a árvore também atende à definição de árvore estritamente binária.

Em resumo, as árvores completas são estritamente binárias, porque a estrutura e organização dos nós em uma árvore completa satisfazem as condições para serem classificadas ambas como árvores completas e estritamente binárias.



unidade



TEMA DE APRENDIZAGEM 7

OPERAÇÕES EM ÁRVORES

MINHAS METAS

- Conhecer técnicas de caminhamento em árvores binárias.
- Aprender a diferenciar os diferentes percursos em árvores binárias.
- Ter contato com técnicas para se balancear uma árvore binária.
- Como criar uma árvore binária formatada especificamente para realizar a busca de elemento.
- Conhecer como manter tal árvore balanceada por meio do algoritmo Adelson-Velskii e Landis (AVL).

INICIE SUA JORNADA

Supondo que você esteja construindo um sistema de gerenciamento de contatos, onde cada contato é representado como um nó em uma árvore binária. Cada nó contém informações como nome, número de telefone e endereço de e-mail. Para tornar o sistema eficiente, você precisa considerar as operações de busca, inserção e remoção de contatos.

Agora, imagine que você tenha uma grande quantidade de contatos e deseja encontrar rapidamente o número de telefone de um contato específico. Além disso, você quer adicionar novos contatos ao sistema de maneira eficiente. **Como resolver estes problemas?**

Neste tema de aprendizagem, veremos como podemos realizar certas operações em árvores binárias para solucionar problemas clássicos. Primeiramente, veremos como é possível gerar diferentes ordens de visitação nos nós de uma árvore por meio das técnicas de caminhamento em árvores binárias, a saber: caminhamento pré-ordem; em-ordem; e pós-ordem.

Em seguida, veremos como podemos criar uma árvore binária formatada especificamente para realizar a busca de elementos. Também teremos uma noção de como podemos manter tal árvore balanceada por meio do algoritmo de Adelson-Velskii e Landis (AVL).

PLAY NO CONHECIMENTO

Agora, quero aproveitar a oportunidade para te convidar a ouvir nosso podcast. Vamos falar sobre como *Explorar os Caminhos: Busca em Árvore Binária e suas Aplicações Práticas*. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

DESENVOLVA SEU POTENCIAL

OPERAÇÕES EM ÁRVORES

Caminhamento em árvores binárias

Em determinadas ocasiões, dependendo dos requisitos de uma aplicação, pode ser preciso percorrer todos os elementos de uma árvore para, por exemplo, exibir

todo o seu conteúdo ao usuário. De acordo com a ordem de visitação dos nós, o usuário pode ter visões distintas de uma mesma árvore.

Imagine que, para percorrer uma árvore, tomemos o nó raiz como nó inicial e, a partir dele, começamos a visitar todos os nós adjacentes a ele para, só então, começar a investigar os outros nós da árvore. Por outro lado, imagine que tomemos um nó folha como ponto de partida e caminhemos em direção à raiz, visitando apenas o ramo da árvore que leva o nó folha à raiz. São maneiras distintas de se visualizar a mesma árvore.

Dito isso, vamos a alguns algoritmos clássicos que ditam as regras de visitação de nós individuais em uma árvore. Ao final da execução de cada um dos algoritmos a seguir, é esperado que sequências de visitação distintas sejam geradas. Para isso, considere que nossas árvores são compostas por registros dinâmicos em C, nos quais cada nó possui ao menos um ponteiro para a subárvore esquerda e outro ponteiro para a subárvore direita.

PERCURSO PRÉ-ORDEM

O **caminhamento pré-ordem**, também conhecido por caminhamento **pré-fixado**, marca, primeiramente, a raiz como visitada, e só depois visitamos as subárvores esquerda e direita, respectivamente.

O Quadro 1, a seguir apresenta um código-fonte no qual a função *preOrdem()* implementa a lógica semântica necessária para fazer com que, a partir do parâmetro raiz, o programa realize o respectivo caminhamento em uma árvore binária.

```
void preOrdem(struct NO* raiz) {
    if(raiz) {
        printf("%d \t", raiz->dado); //visita o nó atual
        preOrdem(raiz->esq);
        preOrdem(raiz->dir);
    }
}
```

Quadro 1 - Programa 2.1 - Função *preOrdem()* / Fonte: Oliveira; Pereira (2019, p. 42).

Considere o caminhamento pré-ordem na árvore da Figura 1, na qual a raiz é o nó F.

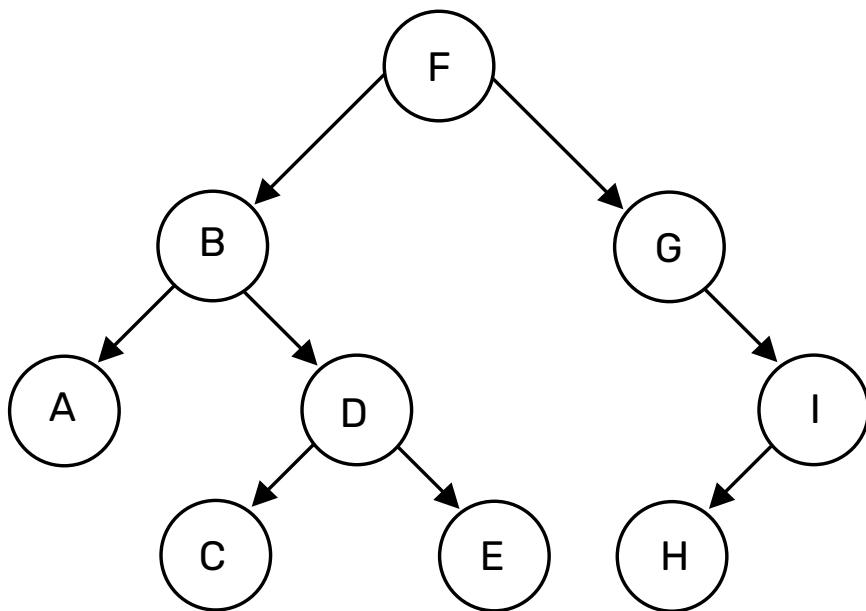


Figura 1 - Árvore binária / Fonte: Oliveira; Pereira (2019, p. 42).

Descrição da Imagem: a figura apresenta uma árvore binária de 9 círculos contendo letras e que estão interligadas com setas. Os círculos de letras interligadas são A - B - F - G - I - H. As letras B - D - E estão interligadas, e apenas o círculo de letra C está interligado no círculo de letra D.

A ordem de visitação produzida pela função *preOrdem()*, levando em conta o nó inicial F, seria a seguinte: **F, B, A, D, C, E, G, I, H**.

PERCURSO EM ORDEM

No caminhamento em ordem, também conhecido por caminhamento **interfazado**, primeiramente, visitamos toda a subárvore esquerda e, só então, a raiz é marcada como visitada. Em seguida, o método em ordem faz a visitação de toda a subárvore direita.

O Quadro 2, a seguir, apresenta um código-fonte no qual a função *emOrdem()* implementa a lógica para fazer com que, a partir da raiz, o programa execute o caminhamento em ordem em uma árvore binária.

```
void emOrdem(NO* raiz) {
    if(raiz) {
        emOrdem(raiz->esq);
        printf("%d \t", raiz->dado); //visita o nó atual
        emOrdem(raiz->dir);
    }
}
```

Quadro 2 - Programa 2.2 - Função *emOrdem()* / Fonte: Oliveira; Pereira (2019, p. 43).

Considere o caminhamento em-ordem na mesma árvore da Figura 1, vista anteriormente, na qual a raiz é o nó F. A ordem de visitação produzida pela função *emOrdem()*, levando em conta o nó inicial F, seria a seguinte: **A, B, C, D, E, F, G, H, I**.

PERCURSO PÓS-ORDEM

O último caminhamento que veremos é o método **pós-ordem**, também conhecido por caminhamento **pós-fixado**. Nesse caso, primeiramente, visitamos toda a subárvore esquerda, depois, toda a subárvore direita. Só após ter visitado as duas **subárvores**, é que marcamos o nó corrente como visitado.

O Quadro 3, a seguir apresenta a função *posOrdem()*, que faz com que, a partir da raiz, o programa execute o caminhamento pós-ordem em uma árvore binária.

```
void posOrdem(NO* raiz){  
    if(raiz){  
        posOrdem(raiz->esq);  
        posOrdem(raiz->dir);  
        printf("%d \t", raiz->dado); //visita o nó  
        atual  
    }  
}
```

Quadro 3 - Programa 2.3 - Função posOrdem() / Fonte: Oliveira; Pereira (2019, p. 44).

Considerando o caminhamento pós-ordem na árvore já conhecida, da Figura 1, a ordem de visitação produzida pela função *posOrdem()*, levando em conta o nó inicial F, seria a seguinte: **A, C, E, D, B, H, I, G, F**.



A **busca em árvores binárias** é uma parte essencial do nosso currículo e é fundamental para muitos algoritmos e estruturas de dados em programação. Para garantir que todos nós tenhamos uma compreensão sólida desse conceito, quero incentivá-lo a assistir ao vídeo sugerido. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

BUSCA EM ÁRVORES BINÁRIAS

Árvores binárias são muito utilizadas para organizar informações na memória devido ao seu grande potencial de busca em um tempo relativamente curto.

Para isso, precisamos criar uma árvore binária de busca. A partir de um vetor, vamos construir uma árvore binária com regras específicas para que ela se torne uma árvore binária de busca.

Uma árvore binária ou é vazia ou tem pelo menos um nó **raiz**. Todo nó tem um **pai** (menos a raiz) e no máximo dois filhos, um **esquerdo** e um **direito**. Tanto o filho esquerdo como o direito podem ser **subárvore**s vazias.

Agora, vamos adicionar uma nova regra para a construção da árvore. O nó raiz será o valor que estiver na posição do meio de um vetor (ordenado ou não). Ao adicionar um novo nó na árvore, verificamos se ele é menor do que a raiz. Caso seja verdade, ele será adicionado na subárvore esquerda, caso contrário, na subárvore direita. Faremos uma simulação usando como base um vetor ordenado representado na Tabela 1.

A primeira linha da figura é o índice do vetor, que vai de 0 a 9. A segunda linha possui os valores de cada posição do vetor.

0	1	2	3	4	5	6	7	8	9
3	1	8	7	20	21	31	40	30	0

Tabela 1 - Vetor para construção de uma árvore binária / Fonte: Oliveira; Pereira (2019, p. 45).

A posição central do vetor é encontrada pela fórmula:

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

Substituindo-se os valores temos:

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

$$meio = \frac{(9)}{2}$$

$$meio = 4,5$$

Como os valores de índices de um vetor são inteiros, **meio** será **4**. O valor do vetor na posição 4 (**vec[4]**) será a raiz da árvore.



Figura 2 - Raiz da árvore / Fonte: Oliveira; Pereira (2019, p. 45).

Descrição da Imagem: a figura apresenta a raiz da árvore um círculo com o número 20 ao centro. Fim da descrição.

Agora que a nossa árvore possui uma raiz, vamos adicionar o primeiro elemento do vetor na árvore. Como **vec[0]** é menor do que a raiz, seu valor será adicionado na subárvore esquerda.

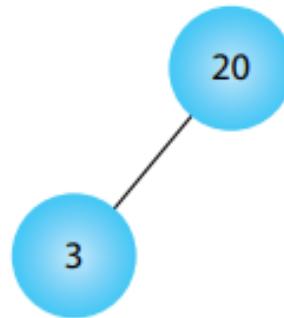


Figura 3 - Subárvore esquerda / Fonte: Oliveira; Pereira (2019, p. 45).

Descrição da Imagem: a figura apresenta subárvore esquerda são dois círculos e estão interligados com uma seta. O círculo superior de número 20 está interligado ao círculo de número três, ele está ao lado esquerdo na diagonal. Fim da descrição.

O próximo valor **vec[1]** também é menor do que a **raiz**, mas a raiz já possui uma subárvore esquerda. Fazemos então uma nova comparação com a raiz da subárvore esquerda. Como **vec[1]** é menor do que a raiz dessa subárvore, ele será adicionado como seu filho esquerdo.

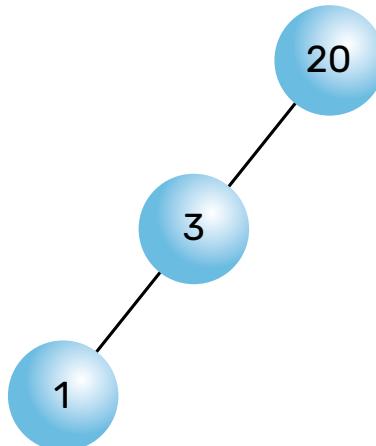


Figura 4 - Subárvore esquerda com filho esquerdo / Fonte: Oliveira; Pereira (2019, p. 45).

Descrição da Imagem: a figura apresenta uma subárvore esquerda com filho esquerdo. São três círculos e estão interligados com uma seta. O círculo superior de número 20 está interligado ao círculo de número 3 que está ao lado esquerdo e abaixo dele ao lado esquerdo na diagonal um círculo com o número um. Fim da descrição.

Dando sequência, vamos verificar o valor contido em **vec[2]**. Seu valor é menor do que **20**, então iremos verificar com o filho esquerdo da raiz. O valor de **vec[2]** é maior do que **3**, então ele entrará na árvore como seu filho direito.

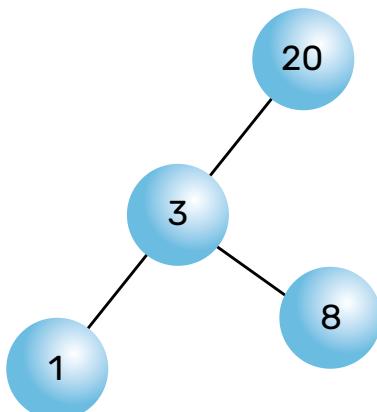


Figura 5 - Subárvore esquerda da raiz com seu filho direito / Fonte: Oliveira; Pereira (2019, p. 45).

Descrição da Imagem: a figura apresenta uma subárvore esquerda da raiz com seu filho direito que são quatro círculos e estão interligados com uma seta. O círculo superior tem o número 20 e abaixo ao lado esquerdo na diagonal um círculo com o número 3 e abaixo dele ao lado esquerdo na diagonal um círculo com o número um. O círculo com o número 3 está interligado na diagonal direita no círculo de número 8. Fim da descrição.

Agora, é a vez de **vec[3]**, que tem valor **7**. Percorremos a árvore em formação respeitando as regras da árvore binária de busca e um novo nó será adicionado como filho esquerdo de **8**.

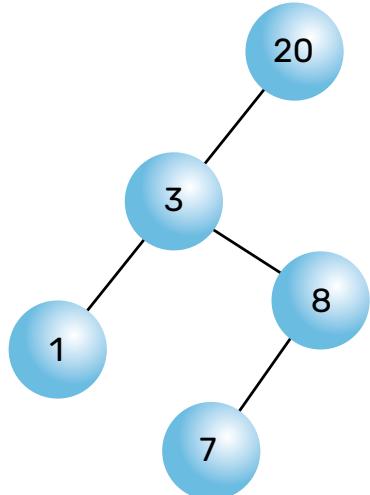


Figura 6 - Subárvore esquerda da raiz com seu filho esquerdo

Fonte: Oliveira; Pereira (2019, p. 45).

Descrição da Imagem: a figura apresenta uma subárvore esquerda da raiz com seu filho direito. São cinco círculos. Os círculos de números 20, três, um estão interligados com uma seta. O círculo superior tem o número 20 e abaixo ao lado esquerdo na diagonal um círculo com o número três e abaixo dele ao lado esquerdo na diagonal um círculo com o número um. O círculo com o número três está interligado na diagonal direita no círculo de número oito. E o círculo de número oito está interligado com uma seta no círculo de número sete.

Fim da descrição.

Chegamos, agora, à metade do vetor da Figura 6. Precisamos de mais cinco interações para finalizar a construção da árvore. A Figura 7 apresenta a árvore final após a inserção dos nós de valores 21, 31, 40, 30 e 0.

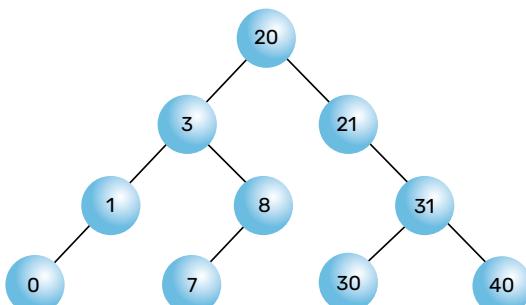


Figura 7 - Árvore Binária de Busca

Fonte: Oliveira; Pereira (2019, p. 47).

Descrição da Imagem: a figura apresenta uma árvore binária de busca. São 10 círculos. Os círculos de números zero, um, três, 20, 21, 31, 40 estão interligados com uma seta. O círculo superior tem o número 20 e abaixo ao lado esquerdo na diagonal um círculo com o número três e abaixo dele ao lado esquerdo na diagonal um círculo com o número um. E abaixo dele ao lado esquerdo na diagonal um círculo com o número zero. O círculo com o número três está interligado na diagonal direita no círculo de número 8. E o círculo de número oito está interligado com uma seta diagonal do lado esquerdo no círculo de número sete. No círculo superior de número 20, abaixo dele do lado direito da diagonal tem um círculo com o número 21 e abaixo dele ao lado direito na diagonal um círculo com o número 31. E em seguida interligado abaixo dele ao lado direito na diagonal um círculo com o número 40. No círculo de número 31 está interligado na diagonal esquerda do círculo de número 30.

Fim da descrição.

Agora, vem o que nos interessa: a realização da busca. Dado um argumento qualquer, se ele for menor do que a **raiz**, ou ele não existe, ou ele se encontra na sua subárvore **esquerda**. Se o valor for maior do que a **raiz**, ou ele não existe ou está na sua subárvore **direita**. A partir da **raiz** vamos descendo pela árvore binária de busca até que o valor seja encontrado ou que encontremos uma **folha** ou uma **subárvore vazia**.

A Figura 8 apresenta uma busca na árvore pelo valor 7. A **raiz** tem valor **20**, então, a busca prossegue na sua subárvore **esquerda**. Como $7 > 3$, a busca continua pela subárvore **direita** do nó **3**. Como $7 < 8$, a busca desce pela subárvore **esquerda** do nó **8** até que o valor 7 seja encontrado.

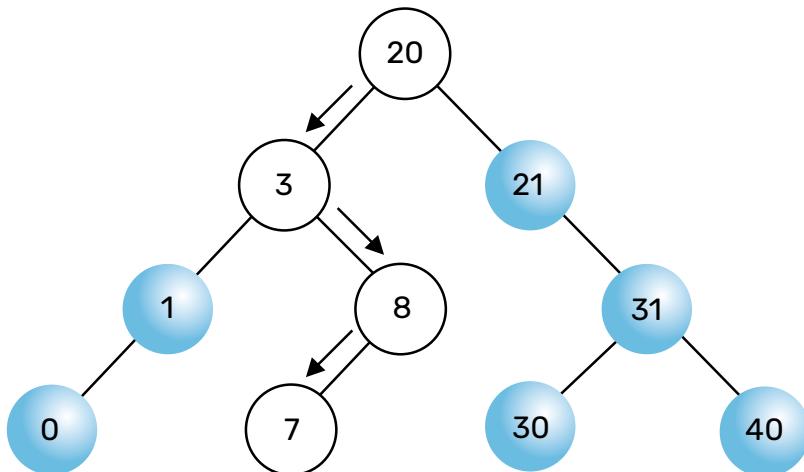


Figura 8 - Busca do valor 7 na árvore binária / Fonte: Oliveira; Pereira (2019, p. 48).

Descrição da Imagem: a figura apresenta uma busca do valor sete na árvore binária. São dez círculos. Os círculos de números zero, um, três, 20, 21, 31, 40 estão interligados com uma seta. O círculo superior tem o número 20 e abaixo ao lado esquerdo na diagonal um círculo com o número três e abaixo dele ao lado esquerdo na diagonal um círculo com o número um. E abaixo dele, ao lado esquerdo na diagonal, um círculo com o número zero. O círculo com o número três está interligado na diagonal direita no círculo de número oito. E o círculo de número oito está interligado com uma seta diagonal do lado esquerdo no círculo de número sete. No círculo superior de número 20, abaixo dele do lado direito da diagonal tem um círculo com o número 21 e abaixo dele ao lado direito na diagonal um círculo com o número 31. E, em seguida, interligado abaixo dele ao lado direito na diagonal um círculo com o número 40. No círculo de número 31, está interligado na diagonal esquerda do círculo de número 30. Há uma seta saindo do círculo 20 para o círculo três que vai para o círculo 8 e para o círculo sete. Estes círculos 20, três, oito e sete estão destacados. Fim da descrição.

Essa técnica se assemelha muito à **busca binária**. A principal diferença é que a árvore binária de busca pode ser implementada dinamicamente na memória, pois não precisamos descobrir onde fica a metade da tabela. Ao invés de dividir o vetor pela metade, ignoramos uma das subárvores para continuar a pesquisa. Esse método é extremamente rápido e eficiente em arquivos com grandes quantidades de registros.

O Programa 2.4 demonstra um algoritmo em linguagem C que encontra um valor por meio de uma árvore binária de busca em um vetor de dados. A função buscaArvoreBinaria() recebe três parâmetros: o vetor **vec** a ser pesquisado, o argumento **arg** a ser encontrado e o tamanho **tam** do vetor.

O laço principal tem duas regras de parada. O laço continua até o valor ser encontrado, que se dá no momento em que a variável **achou** for diferente de -1 ou quando chegar ao final do vetor.

Para ficar mais simples a implementação, estamos utilizando um vetor que guarda os nós de forma ordenada de acordo com o nível de cada nó. Assim, a busca começa na raiz e segue para $2*p+1$ na direção da árvore esquerda ou $2*p+2$ na árvore direita, onde **p** é a posição do nó no vetor. Veja o Quadro 4.



```
//Função de árvore binária de busca
int buscaArvoreBinaria(int vec[], int arg, int tam) {
    int no, achou, fim;
    fim = 0;
    no = 0;
    achou = -1;
    while((achou == -1) && (fim <= tam)) {
        if (arg == vec[no]) {
            achou = no;
        }
        if (arg < vec[no]) {
            no = (2 * no) + 1;
        }
        else {
            no = (2 * no) + 2;
        }
        fim++;
    }
    return(achou);
}
```

Quadro 4 - Programa 2.4 - Árvore de busca binária em C / Fonte: Oliveira; Pereira (2019, p. 49).



INDICAÇÃO DE LIVRO

Estrutura de Dados e Algoritmos em C++

Autor: Adam Drozdek

Sobre o Livro: é muito útil para quem gosta de se aprofundar no tema estrutura de dados. A partir da obra, o leitor é capaz de ter contato com as mais simples estruturas de dados até as mais elaboradas, tendo exemplos visuais e também com código-fonte em linguagem C++.



ÁRVORES AVL

A forma como os elementos são inseridos em uma árvore binária de busca pode fazer com que a busca se torne altamente ineficiente. Considere o conjunto de dados a seguir:

2 - 5 - 10 - 17 - 25 - 32

Se inserirmos tais dados, do primeiro até o último, em uma árvore de busca binária, teremos a árvore estruturada da seguinte forma:

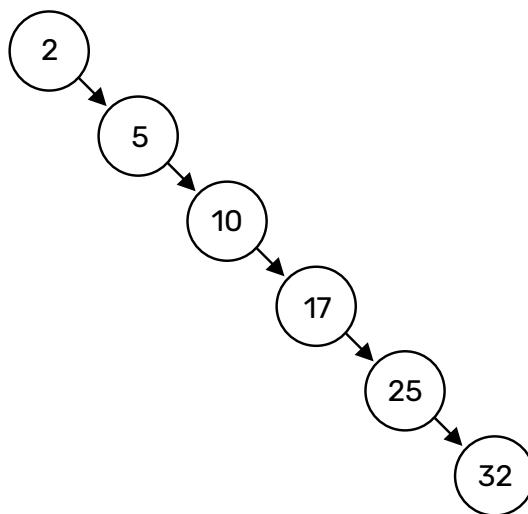


Figura 9 - Árvore de busca binária desbalanceada
Fonte: Oliveira; Pereira (2019, p. 50).

Descrição da Imagem: a figura apresenta uma árvore de busca binária desbalanceada. São seis círculos. Os círculos de números dois, cinco, dez, 17, 25, 32 estão interligados com uma seta na diagonal direita. O círculo superior tem o número dois e abaixo dele ao lado direito na diagonal um círculo com o número cinco e abaixo dele ao lado direito na diagonal um círculo com o número dez, em seguida abaixo dele ao lado direito na diagonal um círculo com o número 17 e abaixo dele ao lado direito na diagonal um círculo com o número 25 e por último abaixo dele ao lado direito na diagonal um círculo com o número 32. Fim da descrição.

Repare que a estrutura de dados se assemelha muito mais a uma lista linear encadeada, do que com uma árvore binária de busca. Em consequência, operações de busca nessa estrutura de dados serão feitas de maneira sequencial, em vez de aproveitarem a capacidade de ignorar subárvore, cortando caminho na busca. Dizemos que a árvore da Figura 9 encontra-se **desbalanceada**.

A partir daí, surge o conceito de balanceamento. Dizemos que uma árvore balanceada tende a manter sua altura tão pequena quanto possível, à medida que são realizadas novas inserções ou remoções de dados.

Considerando o mesmo conjunto de dados que vimos anteriormente, poderíamos tentar balancear a árvore binária de busca, resultando em algo parecido com o seguinte:

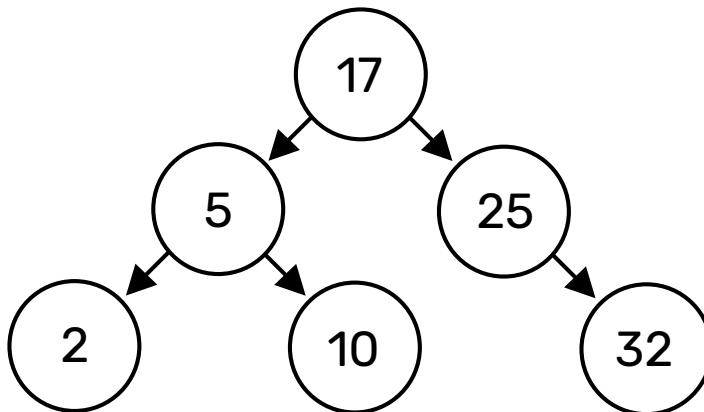


Figura 1.0 - Árvore de busca binária balanceada / Fonte: Oliveira; Pereira (2019, p. 50).

Descrição da Imagem: a figura apresenta uma árvore de busca binária desbalanceada. São 6 círculos. Os círculos de números dois, cinco, dez, 17, 25, 32 estão interligados com uma seta. Os círculos dois, cinco, 17 estão interligados na diagonal esquerda e a partir do círculo 17, 25 e 32 estão interligados na diagonal direita. O círculo de número cinco está interligado na diagonal direita do círculo dez. Fim da descrição.

O **conceito de balanceamento** está relacionado à altura das subárvore que compõem uma árvore binária. É importante lembrar que a altura (ou profundidade) de uma subárvore é igual ao número de nós visitados desde a raiz até o nó folha mais distante (uma subárvore vazia possui altura -1, por definição). Dizemos que um nó está平衡ado caso o valor absoluto da diferença entre as alturas das subárvore esquerda e direita seja menor ou igual a 1. Chamaremos de fator de balanceamento (F_b) o resultado da diferença entre a altura da subárvore esquerda (H_e) de um nó pela altura da subárvore direita (H_d) do mesmo nó, de acordo com a seguinte fórmula:

$$Fb = He - Hd$$

Vamos a alguns exemplos. A árvore da Figura 11 encontra-se balanceada. Nessa árvore, ilustramos a altura (H) de cada subárvore na porção superior esquerda de cada nó, e o respectivo fator de平衡amento (Fb) no canto superior direito de cada nó. Além disso, as raízes de subárvores vazias estão destacadas como quadrados pontilhados. Repare como cada fator de平衡amento (destacados ao lado superior direito de cada nó) é o resultado do valor da altura da subárvore esquerda menos a altura da subárvore direita.

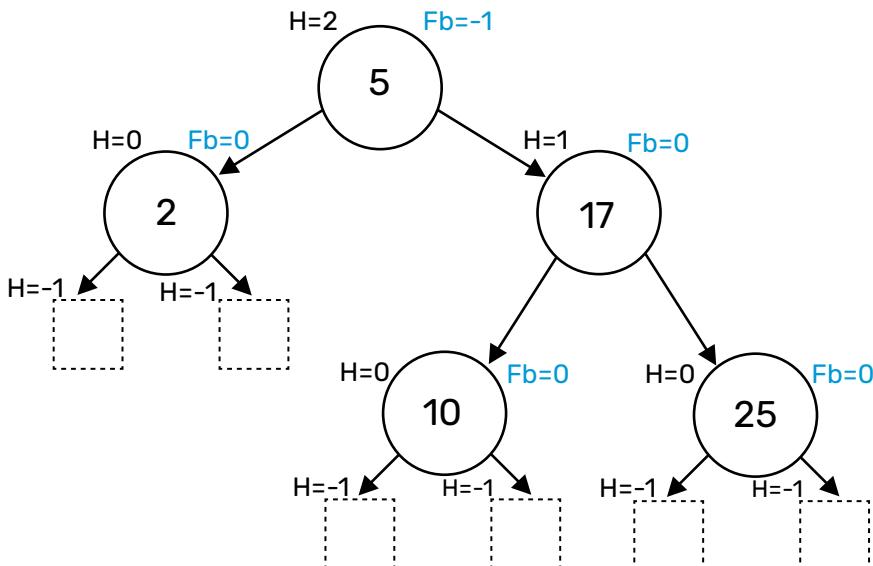


Figura 11 - Fatores de平衡amento com todos os nós balanceados / Fonte: Oliveira; Pereira (2019, p. 51).

Descrição da Imagem: a figura apresenta os fatores de平衡amento com todos os nós balanceados. São 6 círculos. Os círculos de números dois, cinco, dez, 17, 25 estão interligados com uma seta. Os círculos dois, cinco estão interligados na diagonal esquerda e a partir do círculo cinco, 17, 25 estão interligados na diagonal direita. O círculo de número 17 está interligado na diagonal esquerda do círculo dez. No círculo de número 2 tem duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada. Cada quadrado há uma enumeração H igual menos um. Nos círculos de número 10 e 25 também têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com H igual menos um. Em torno dos outros círculos tem-se H=0, Fb=0, H=2, Fb=1, H=1. Fim da descrição.

Todavia, existem situações nas quais uma árvore de busca possui nós desbalanceados. Por exemplo, imagine que adicionemos um nó cujo valor é igual a 32. Seguindo as regras de inserção para uma árvore binária de busca, tal nó seria adicionado como filho à direita do nó de valor igual a 25. Com isso, temos uma nova árvore, como ilustra a Figura 12.

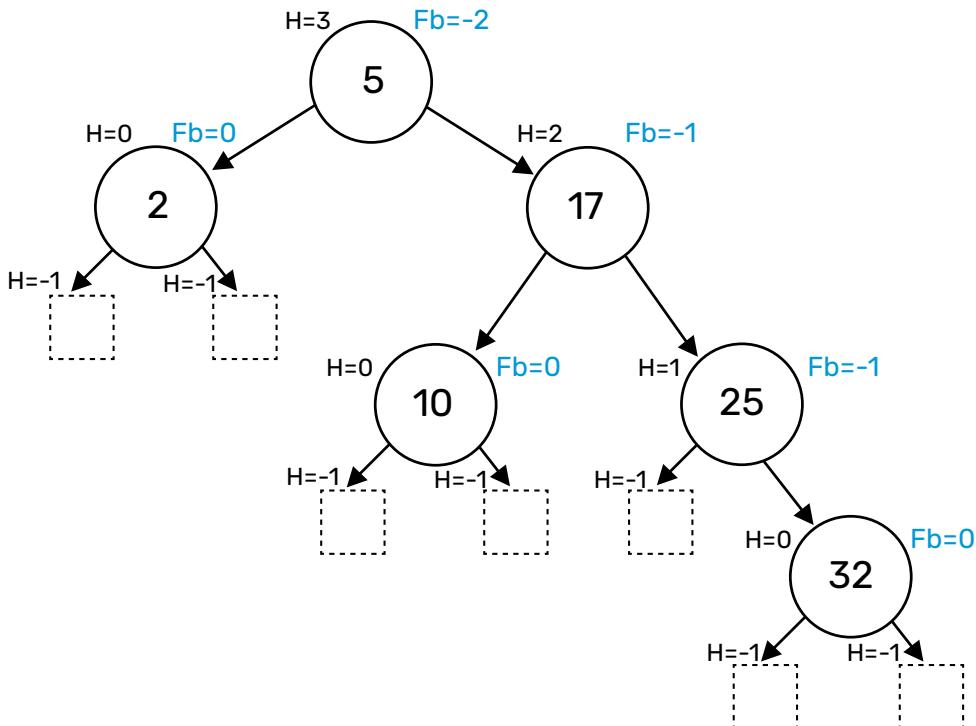


Figura 12 - Fatores de balanceamento com um nó desbalanceado / Fonte: Oliveira; Pereira (2019, p. 52).

Descrição da Imagem: a figura apresenta os fatores de平衡amento com um nó desbalanceado. São 6 círculos. Os círculos de números 2, 5, 10, 17, 25, 32 estão interligados com uma seta. Os círculos 2, 5 estão interligados na diagonal esquerda e a partir do círculo 5, 17, 25, 32 estão interligados na diagonal direita. O círculo de número 17 está interligado na diagonal esquerda do círculo 10. No círculo de número 2 tem duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada. Cada quadrado há uma enumeração H = -1. No círculo de número 10 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com H = -1. No círculo de número 25 tem uma seta virada para o lado diagonal esquerdo e um quadrado com H = -1. No círculo de número 32 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com H = -1. Em torno dos outros círculos tem-se H=0, Fb=0, H=2, Fb=1, H=1. Fim da descrição.

Analisando a Figura 12, notamos que a inserção do nó de valor 32 fez com que a raiz da árvore perdesse seu balanceamento. Note que a altura da subárvore esquerda, em relação à raiz, é igual a 0, porém a altura da subárvore direita é igual a 2. Com isso, o fator de平衡amento da raiz é igual a -2 que, em valor absoluto, é maior do que 1. Assim, pela definição, dizemos que o nó de valor igual a 5 encontra-se desbalanceado.

**dizemos que o
nó de valor igual
a 5 encontra-se
desbalanceado**

ZOOM NO CONHECIMENTO

Para resolver o problema do desbalanceamento de árvores binárias de busca, os pesquisadores Adelson-Velskii e Landis, em 1962 , criaram um algoritmo que leva as iniciais de seus nomes. As árvores AVL são, nesse sentido, árvores nas quais todos os nós encontram-se balanceados. Para cada nó, a diferença entre as alturas de suas subárvores não pode ser igual ou superior a 2, em valor absoluto.

Como vimos, uma árvore balanceada pode perder essa característica quando um novo elemento é inserido. Isso também pode ocorrer quando um elemento é removido da árvore binária. Assim, quando ocorrem operações de inserção ou remoção em árvores AVL, recalcula-se os fatores de balanceamento de cada nó para, assim, poder executar rotações nos nós problemáticos, na tentativa de restabelecer seu balanceamento. Existem, basicamente, quatro tipos de rotações: dois tipos de rotações simples e dois tipos de rotações duplas.

APROFUNDANDO

De acordo com Adam Drozdek (2008), quando uma árvore AVL se desbalanceia inserindo-se um nó na subárvore que se encontra à direita do filho direito, é necessário realizar uma rotação simples. Simetricamente, se inserirmos um nó na subárvore esquerda do filho esquerdo, é preciso realizar uma rotação simples no sentido oposto ao anterior.

Dessa forma, podemos observar que, no exemplo da Figura 12, foi realizada uma inserção na subárvore direita do filho direito em relação ao nó desbalanceado, ou seja, o nó 32 foi inserido na subárvore direita do nó 17, que é filho à direita do nó 5.

Para tornar a árvore balanceada, precisamos realizar uma rotação, conforme o ilustrado na Figura 13.

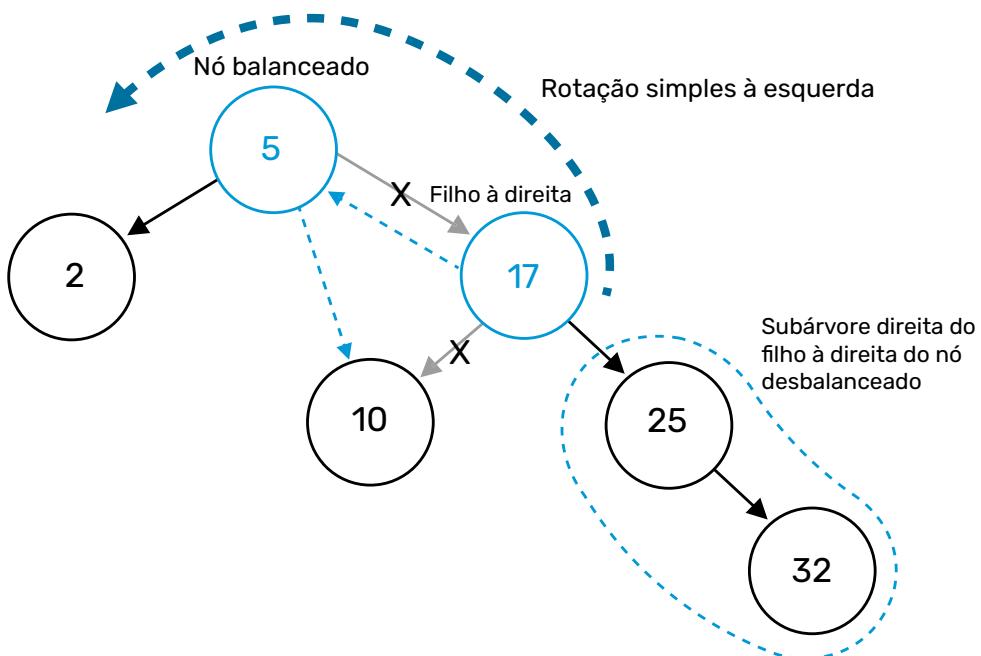


Figura 13 – Corrigindo uma inserção na subárvore direita do filho à direita com rotação simples à esquerda
Fonte: Oliveira; Pereira (2019, p. 53).

Descrição da Imagem: a figura apresenta a correção de uma inserção na subárvore direita do filho à direita com rotação simples à esquerda. São 6 círculos. Os círculos de números 2, 5, 10, 17, 25, 32 estão interligados com uma seta. Os círculos 2, 5 estão interligados na diagonal esquerda e a partir do círculo 5, 17, 25, 32 estão interligados na diagonal direita. O círculo de número 5 está interligado na diagonal direita do círculo 10. O círculo de número 17 está interligado na diagonal esquerda do círculo 5. O círculo de número 25 tem uma seta virada para o lado diagonal direito para o círculo 32. Nesta figura apresenta um nó desbalanceado, filho à direita e uma rotação simples à esquerda no grupo de círculos 17 e 5. Nos círculos 25 e 32 uma subárvore direita do filho à direita do nó desbalanceado. Fim da descrição.

Para realizarmos a rotação à esquerda, precisamos “puxar” o nó 17 para cima, fazendo com que o nó 5 passe a ser o filho à esquerda do nó 17. Além disso, a subárvore com raiz no nó 10 é “deserdada” pelo nó 17, e o nó 5 a “adota”, fazendo com que o nó 10 passe a ser o filho à direita do nó 5, deixando de ser o filho à

esquerda do nó 17. Dessa forma, após realizada a rotação, devemos recalcular a altura de cada subárvore, bem como os fatores de平衡amento de cada nó. Nesse caso, a árvore balanceada resultante pode ser visualizada na Figura 14.

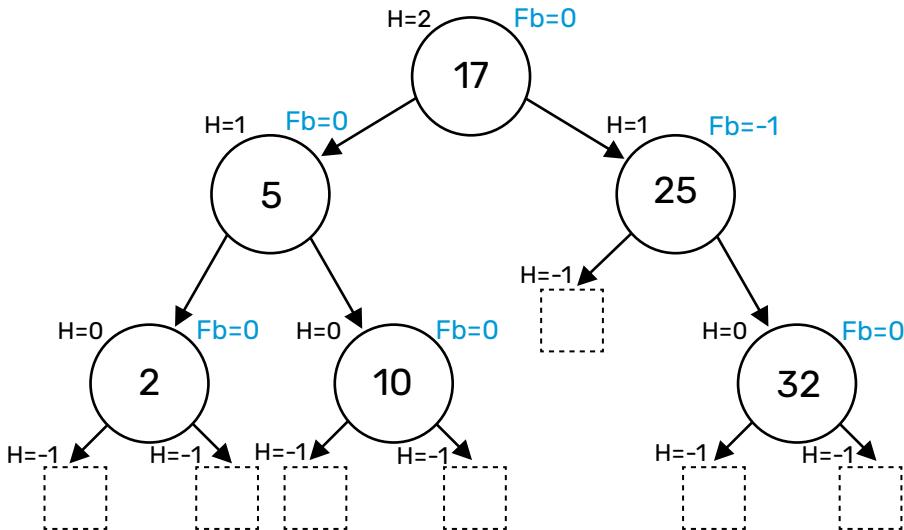


Figura 14 - Árvore AVL balanceada após uma inserção / Fonte: Oliveira; Pereira (2019, p. 54).

Descrição da Imagem: a figura apresenta a árvore AVL balanceada após uma inserção. São 6 círculos. Os círculos de números 2, 5, 10, 17, 25, 32 estão interligados com uma seta. Os círculos 2, 5, 17 estão interligados na diagonal esquerda e a partir do círculo 17, 25, 32 estão interligados na diagonal direita. O círculo de número 5 está interligado na diagonal direita do círculo 10. No círculo de número 2 e tem duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada. Cada quadrado há uma enumeração $H = -1$. No círculo de número 10 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H = -1$. No círculo de número 25 tem uma seta virada para o lado diagonal esquerdo e um quadrado com $H = -1$. No círculo de número 32 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H = -1$. Em torno dos outros círculos tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. Fim da descrição.

A partir dessa nova árvore balanceada, adicionaremos um nó de valor igual a 1. Para isso, devemos continuar seguindo a regra de inserção em árvore de busca binária. Por isso, temos de percorrer da raiz 17 em direção a uma folha, até poder inserir o 1 à esquerda do nó 2. Dessa forma, temos uma nova árvore, com novas alturas e fatores de balanceamento, como podemos observar na Figura 15. Repare que a árvore não está desbalanceada em nó algum.

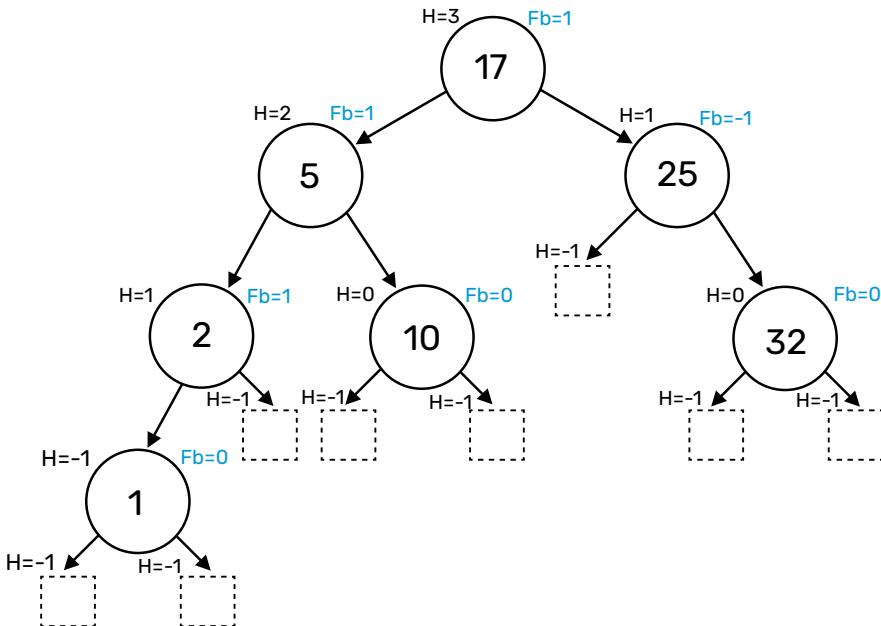


Figura 15 - Inserindo o nó 1 / Fonte: Oliveira; Pereira (2019, p. 54).

Descrição da Imagem: : a figura apresenta a inserindo o nó 1. São 7 círculos. Os círculos de números 1, 2, 5, 10, 17, 25, 32 estão interligados com uma seta. Os círculos 1, 2, 5, 17 estão interligados na diagonal esquerda e a partir do círculo 17, 25, 32 estão interligados na diagonal direita. O círculo de número 5 está interligado na diagonal direita do círculo 10. O círculo de número 2 está interligada no círculo de número 1. No círculo 1 há duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada. Cada quadrado há uma enumeração $H=-1$. Continuando no círculo 2, outra seta sai na diagonal da direita e aponta para um quadrado $H=-1$. No círculo de número 10 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H = -1$. No círculo de número 25 tem uma seta virada para o lado diagonal esquerdo e um quadrado com $H = -1$. No círculo de número 32 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H = -1$. Em torno dos outros círculos tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. Fim da descrição.

Sempre que inserimos um nó, devemos checar se todos os nós percorridos desde a raiz até o nó folha estão平衡ados. Como podemos observar na Figura 15, os nós percorridos foram, nesta ordem: 17, 5 e 2. É possível reparar que, após a inserção, foram recalculados os fatores de balanceamento Fb de tais nós e nenhum deles se tornou desbalanceado. Repare, ainda, que somente

Nós percorridos devem ser checados à procura de desbalanceamentos

os nós 17, 5 e 2 tiveram seus fatores de balanceamento alterados, ou seja, isso reforça o fato de que apenas os nós percorridos devem ser checados à procura de desbalanceamentos.

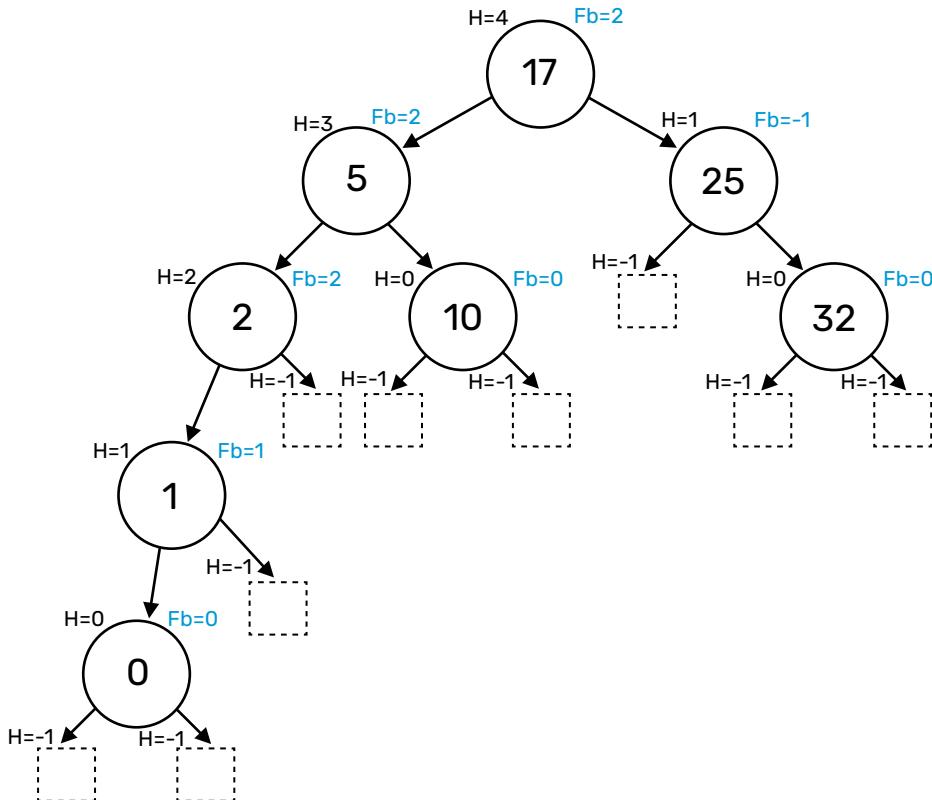


Figura 16 - Inserindo o nó 0 / Fonte: Oliveira; Pereira (2019, p. 55).

Descrição da Imagem: a figura apresenta a inserindo o nó 1. São 8 círculos. Os círculos de números 0, 1, 2, 5, 10, 17, 25, 32 estão interligados com uma seta. Os círculos 0, 1, 2, 5, 17 estão interligados na diagonal esquerda e a partir do círculo 17, 25, 32 estão interligados na diagonal direita. O círculo de número 5 está interligado na diagonal direita do círculo 10. O círculo de número 2 está interligada no círculo de número 1 e na diagonal da direita a seta aponta para o quadrado $H=-1$. O círculo de número 1 está interligado ao círculo de número 0 e na diagonal da direita a seta aponta para o quadrado $H=-1$. No círculo 0 há duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada. Cada quadrado há uma enumeração $H=-1$. No círculo de número 10 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H = -1$. No círculo de número 25 tem uma seta virada para o lado diagonal esquerdo e um quadrado com $H = -1$. No círculo de número 32 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H = -1$. Em torno dos outros círculos tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. Fim da descrição.

Note, agora, que, na Figura 16, temos a inserção do nó de valor igual a 0. Para adicionar o nó 0, tivemos de percorrer o nó 17, o 5, o 2 e o 1, para, finalmente, posicionar o 0 à esquerda do nó 1. Ao recalcular os fatores de balanceamento, chegamos à conclusão de que os nós 2, 5 e 17 estão desbalanceados. A partir dessas informações, temos condições de tentar consertar nossa árvore, de baixo para cima.

Novamente, devemos dar atenção ao fato de que apenas os nós percorridos durante a inserção tiveram chance de se tornar desbalanceados, pois somente suas subárvores tiveram as respectivas alturas alteradas após a inserção.

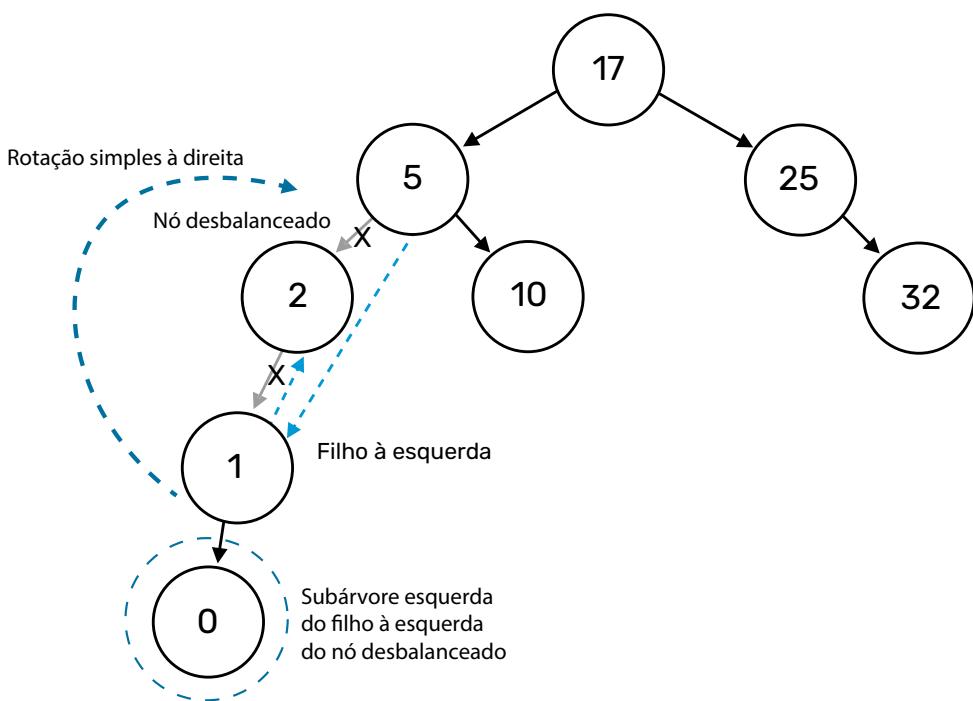


Figura 17 - Corrigindo uma inserção na subárvore esquerda do filho à esquerda com rotação simples à direita
Fonte: Oliveira; Pereira (2019, p. 56).

Descrição da Imagem: a figura apresenta uma correção de uma inserção na subárvore esquerda do filho à esquerda com rotação simples à direita. São 8 círculos. Os círculos de números 0, 1, 2, 5, 10, 17, 25, 32 estão interligados com uma seta. Os círculos 0, 1, 2, 5, 17 estão interligados na diagonal esquerda e a partir do círculo 17, 25, 32 estão interligados na diagonal direita. O círculo de número 5 está interligado na diagonal direita do círculo 10. Nesta figura apresenta um nó desbalanceado, filho à esquerda, uma rotação simples à direita no grupo de círculos 5 e 2. E no círculo zero uma subárvore esquerda do filho à esquerda do nó desbalanceado. Fim da descrição.

De maneira simétrica à situação que levou à rotação feita anteriormente, temos o caso oposto. Agora, realizamos a inserção de um nó na subárvore esquerda do filho à esquerda do nó desbalanceado. Para resolver esse desbalanceamento, devemos realizar uma rotação simples para a direita, fazendo com que o nó 5 passe a ser o pai do nó 1 que, por sua vez, passa a ser o filho à esquerda do nó 5. O nó 2 deixa de ser filho do nó 5 e passa a ser o filho à direita do nó 1. Caso o nó 1 tenha alguma subárvore à direita, a raiz dessa subárvore passa a ser filha à esquerda do nó 2 (nó 2 “adota” o filho órfão do nó 1). O resultado dessa rotação pode ser observado na Figura 18.

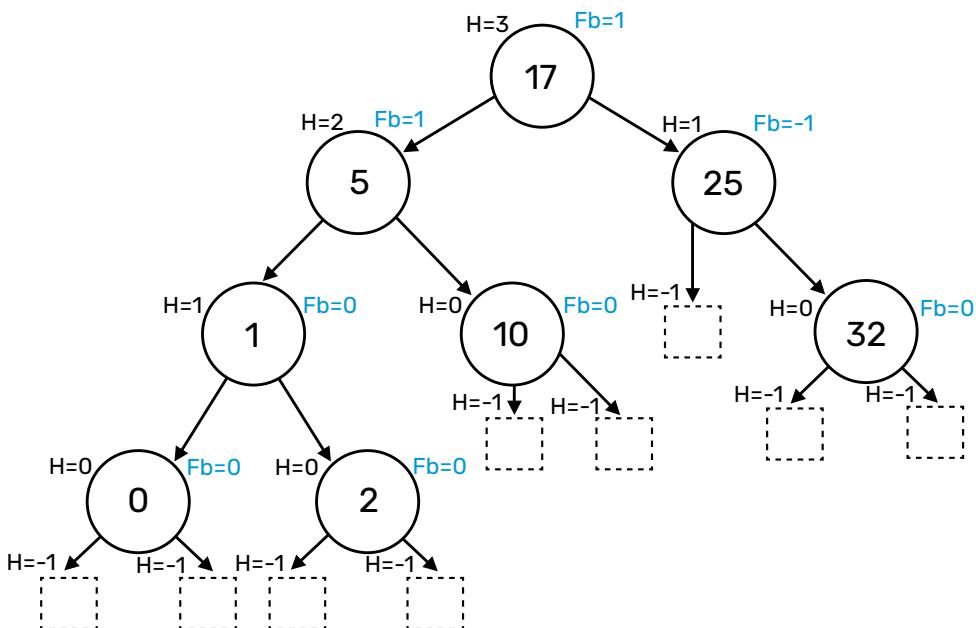


Figura 18 – Balanceamento de árvore AVL após rotação simples à direita / Fonte: Oliveira; Pereira (2019, p. 56).

Descrição da Imagem: a figura apresenta um balanceamento de árvore AVL após rotação simples à direita. São 8 círculos. Os círculos de números 0, 1, 2, 5, 10, 17, 25, 32 estão interligados com uma seta. Os círculos 0, 1, 5, 17 estão interligados na diagonal esquerda e a partir do círculo 17, 25, 32 estão interligados na diagonal direita. O círculo de número 5 está interligado na diagonal direita do círculo 10. No círculo de número 10 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H=-1$. O círculo de número 1 está interligado na diagonal direita do círculo 2. No círculo de número 2 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H=-1$. No círculo de número zero e tem duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada e com $H=-1$. No círculo de número 25 tem uma seta virada para o lado diagonal esquerdo e um quadrado com $H = -1$. No círculo de número 32 têm duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H = -1$. Em torno dos outros círculos tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. Fim da descrição.

A Figura 18 nos mostra que, realizando uma rotação simples no primeiro nó desbalanceado, partindo do nó recém-inserido em direção à raiz, balanceou-se toda a árvore.

Todavia, em situações nas quais os nós ascendentes continuam desbalanceados após uma rotação, é preciso continuar corrigindo a árvore, até que todos os nós da folha até a raiz sejam平衡ados.

Agora, considere outra árvore, na qual os nós 17, 32 e 25 foram inseridos na árvore de busca binária, na respectiva ordem. Com isso, temos a raiz 17 se tornando desbalanceada, como podemos ver na Figura 19. Contudo, a quebra no balanço foi realizada por meio de uma inserção na subárvore esquerda do filho direito em relação ao nó desbalanceado, ou seja, o número 25 foi inserido à esquerda do nó 32 que, por sua vez, está à direita do nó 17.

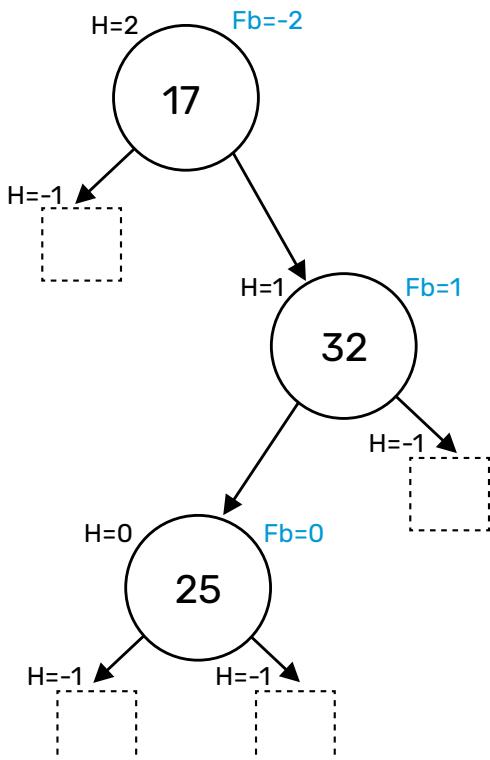


Figura 19 - Inserção na subárvore esquerda do filho à direita do nó desbalanceado
Fonte: Oliveira; Pereira (2019, p. 57).

Descrição da Imagem: a figura apresenta uma inserção na subárvore esquerda do filho à direita do nó desbalanceado. São 3 círculos: 17, 32 e 25. O círculo 17 tem uma seta na diagonal esquerda apontada para o quadrado $H=-1$. A outra seta na diagonal direita está interligada no círculo 32 e este tem uma seta na diagonal direita apontada para o quadrado $H=-1$. A outra seta na diagonal esquerda está interligada no círculo 25 e este tem duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H=-1$. Em torno dos outros círculos tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. Fim da descrição.

Em uma situação como essas, uma única rotação simples em torno do nó desbalanceado não resolve o problema. Isso ocorre, pois, ao tentar realizar uma única rotação à esquerda, continuaríamos tendo o nó raiz, 17, desbalanceado. Nessas situações, é preciso empregar a rotação dupla direita-esquerda, que é feita em duas etapas.

Nesse caso, primeiramente, rotaciona-se à direita a subárvore com raiz em 25, em direção ao nó 32 (Figura 20 (a) e (b)); num segundo momento, rotacionamos à esquerda a subárvore com raiz em 25, em direção ao nó 17 (Figura 20 (c) e (d)), como podemos ver na Figura 20.

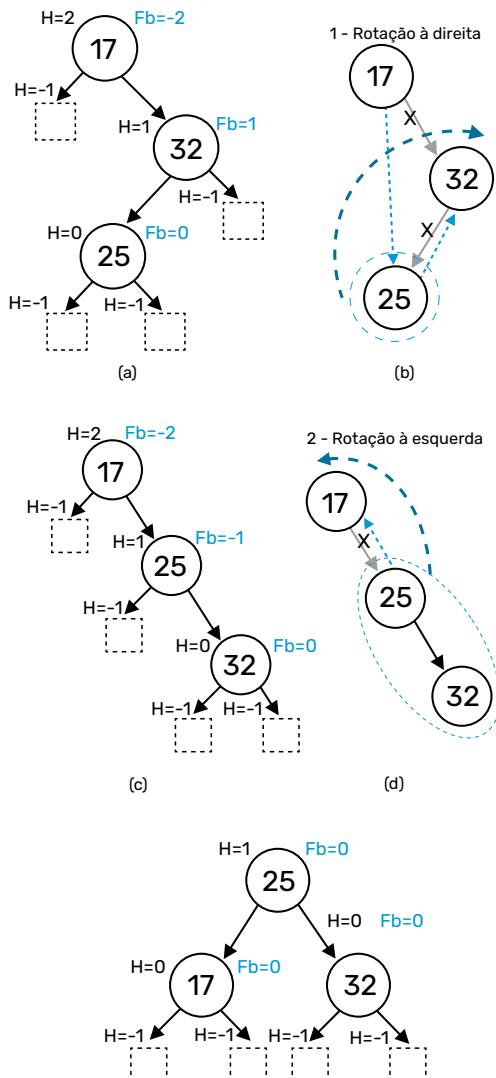


Figura 20 - Corrigindo uma inserção na subárvore esquerda do filho à direita com rotação dupla esquerda/direita
Fonte: Oliveira; Pereira (2019, p. 58).

Descrição da Imagem: a figura apresenta uma correção de uma inserção na subárvore esquerda do filho à direita com rotação dupla esquerda/direita. São 5 grupos de figuras (a, b, c, d, e). A figura 20 (a) apresenta os 3 círculos: 17, 32 e 25. O círculo 17 tem uma seta na diagonal esquerda apontada para o quadrado $H=-1$. A outra seta na diagonal direita está interligada no círculo 32 e este tem uma seta na diagonal direita apontada para o quadrado $H=-1$. A outra seta na diagonal esquerda está interligada no círculo 25 e este tem duas setas cada uma virada para o lado diagonal esquerdo e direito e apontadas para um quadrado cada com $H=-1$. Em torno dos outros círculos tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. A figura 20 (b) com os 3 círculos: 17, 32 e 25 em rotação à direita. A figura 20 (c) são os 3 círculos: 17, 25 e 32 interligados na diagonal direita e outras setas na diagonal esquerda apontam para o quadrado $H=-1$. Na figura 20 (d) com os 3 círculos: 17, 32 e 25 em rotação à esquerda. E na figura 20 (e) apresenta o círculo 17 interligado ao círculo 25 na diagonal esquerda e o círculo 25 interligado no círculo 32 na diagonal direita. Os círculos 17 e 32 apresentam duas setas na diagonal esquerda e na diagonal direita apontadas para o quadrado cada um com $H=-1$. Em torno dos outros círculos tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. Fim da descrição.

Por outro lado, temos uma **situação simétrica** à que foi recém-apresentada, que também é resolvida com uma rotação dupla. Considere, agora, que inserimos os nós 10 e 13, um em seguida do outro, levando à situação ilustrada pela Figura 21.

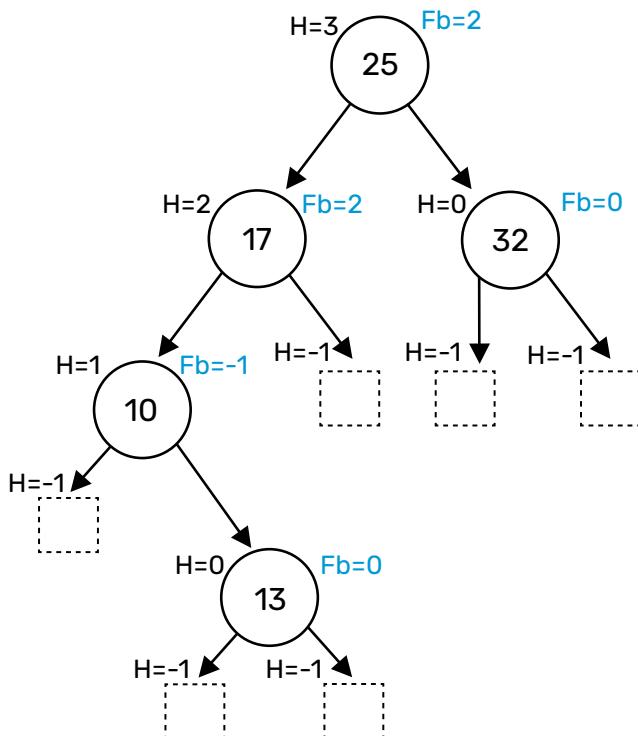


Figura 21 - Inserção na subárvore à direita do filho à esquerda
Fonte: Oliveira; Pereira (2019, p. 59).

Descrição da Imagem: a figura apresenta uma inserção na subárvore à direita do filho à esquerda. São 5 círculos: 13, 10, 17, 25, 32. Do círculo 25, sai uma seta na diagonal esquerda para o círculo 17. Do círculo 17 sai uma seta na diagonal esquerda para o círculo 10 e na diagonal da direita sai uma para o círculo 13. Do círculo 25 sai outra seta na diagonal da direita e vai para o círculo 32. Do círculo 13 e 32, saem duas setas na diagonal esquerda e na diagonal direita apontadas para o quadrado cada um com $H=-1$. Do círculo 10, sai uma seta na diagonal esquerda para o $H=-1$. Do círculo 17, sai uma seta na diagonal direita para o $H=-1$. Em torno dos outros círculos, tem-se $H=0, Fb=0, H=2, Fb=1, H=1$. Fim da descrição.

Após a inserção do 10 à esquerda do 17, e do 13 à direita do 10, temos uma situação na qual os fatores de平衡amento dos nós 17 e 25 quebram a regra da árvore AVL. Nesse caso, temos uma inserção na subárvore direita do filho

esquerdo em relação ao nó desbalanceado. Assim, precisamos nesse caso, realizar rotação dupla esquerda-direita, em duas etapas. Primeiramente, fazemos com que o nó 10 se torne o filho à esquerda do nó 13, como ilustrado na Figura 22.

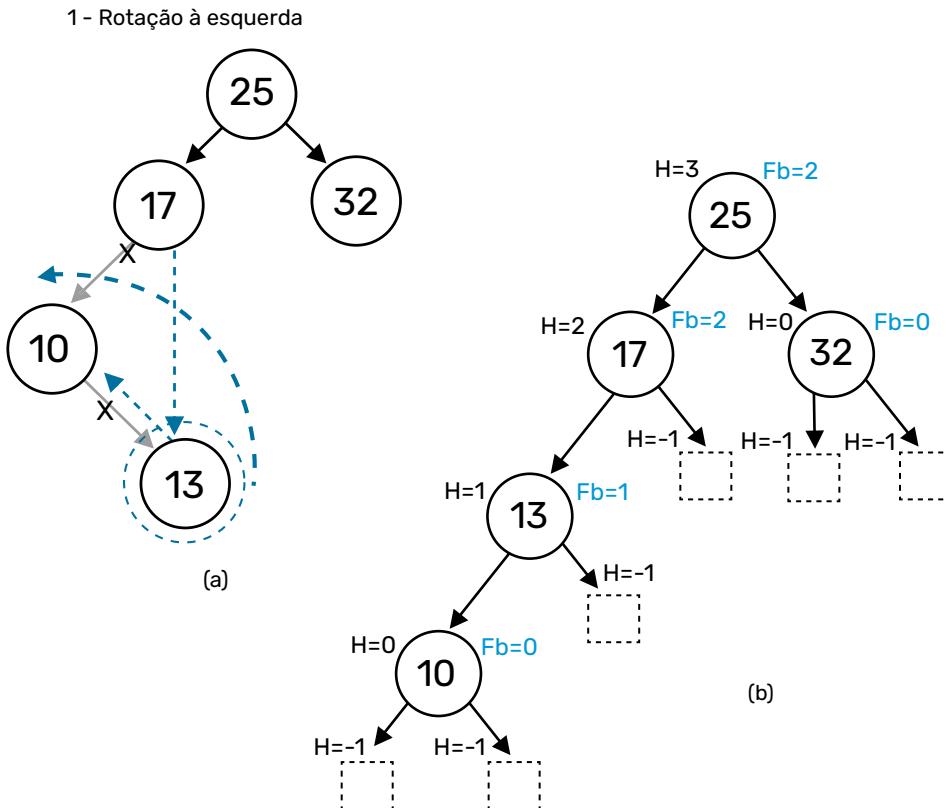


Figura 22 – Rotação dupla - 1^a etapa: rotação à esquerda
Fonte: Oliveira; Pereira (2019, p. 60).

Descrição da Imagem: a figura apresenta uma rotação dupla - 1^a etapa: rotação à esquerda. São 2 grupos de figuras (a, b). A figura 22 (a) apresenta 5 círculos: 13, 10, 17, 25 e 32. Nesta figura apresenta-se a rotação à esquerda. Na figura 22 (b) temos 5 círculos: 10, 13, 17, 25, 32. Do círculo 25 sai uma seta na diagonal esquerda para o círculo 17. Do círculo 17 sai uma seta na diagonal esquerda para o círculo 13. Do círculo 13 sai uma seta na diagonal esquerda para o círculo 10. Do círculo 10 saem duas setas na diagonal esquerda e na diagonal direita apontadas para o quadrado cada um com $H=-1$. Do círculo 25 sai outra seta na diagonal da direita e vai para o círculo 32. No círculo 13 e 17 saem setas na diagonal direita apontadas para o quadrado cada um com $H=-1$. E nos círculos 10 e 32 saem duas setas uma na diagonal esquerda e outra para a direita apontando para o $H=-1$. Em torno dos outros círculos tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. Fim da descrição.

Depois, para balancear o nó 17, realizamos uma rotação à direita a partir do nó 13, como podemos visualizar na Figura 23.

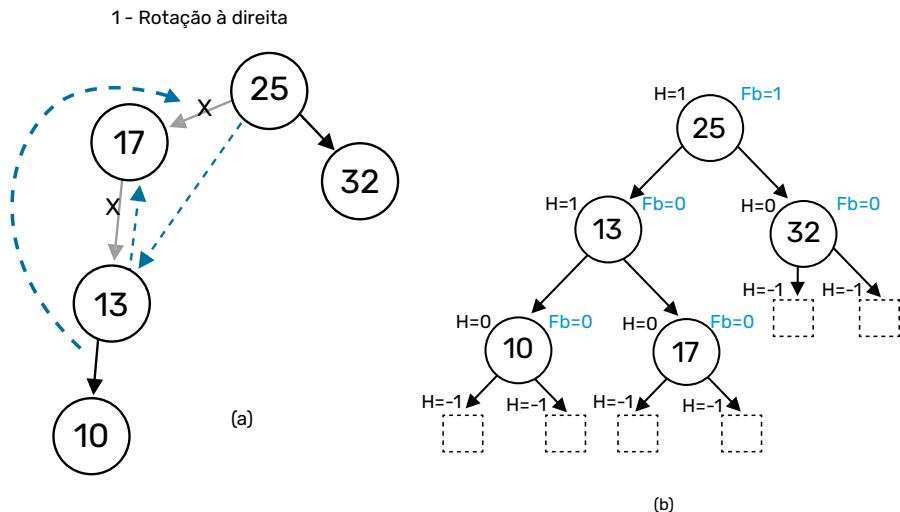


Figura 23 - Rotação dupla - 2^a etapa: rotação à direita
Fonte: Oliveira; Pereira (2019, p. 60).

Descrição da Imagem: a figura apresenta uma rotação dupla - 2^a etapa: rotação à direita. São dois grupos de figuras (a, b). A figura 23 (a) apresenta 5 círculos: 10, 13, 17, 25 e 32. Nesta figura, apresenta-se a rotação à esquerda. Na figura 22 (b), temos 5 círculos: 10, 13, 17, 25, 32. Do círculo 25, sai uma seta na diagonal esquerda para o círculo 13 e outra seta na diagonal direita para o círculo 32. Do círculo 13, sai uma seta na diagonal esquerda para o círculo 10 e outra seta na diagonal direita para o círculo 17. Do círculos 10, 17 e 32, saem setas na diagonal direita apontadas para o quadrado cada um com $H=-1$. Em torno dos outros círculos, tem-se $H=0$, $Fb=0$, $H=2$, $Fb=1$, $H=1$. Fim da descrição.

EU INDICO

Estudante, acesse o artigo sugerido para conhecer mais e se aprofundar em AVL balanceada. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

Com isso, temos nossa árvore AVL balanceada novamente. Basta observar os fatores de平衡amento do resultado final.

Dessa forma, abordamos as quatro situações nas quais precisamos balancear nossa árvore AVL, a saber: duas rotações simples e duas rotações duplas. Podemos resumir as quatro situações que exigem as respectivas rotações com o:

5. **Inserção na subárvore direita do filho à direita:** solução com rotação simples à esquerda.
6. **Inserção na subárvore esquerda do filho à esquerda:** solução com rotação simples à direita.
7. **Inserção na subárvore esquerda do filho à direita:** solução com rotação dupla direita-esquerda.
8. **Inserção na subárvore direita do filho à esquerda:** solução com rotação dupla esquerda-direita.

Em suma, uma árvore AVL é uma árvore binária de busca na qual o fator de balanceamento de cada um de seus nós não pode ser maior que 1 ou menor que -1. Caso o fator de balanceamento de um ou mais nós, seja maior ou igual a 2, em valores absolutos, é preciso analisar a ordem das inserções para aplicar a rotação mais adequada para corrigir o balanceamento de cada um dos nós. Isso faz com que buscas em uma árvore AVL sejam mais eficientes.

EU INDICO

Compreendendo melhor as árvores AVL

As árvores AVL, nomeadas em homenagem a seus inventores, Adelson-Velsky e Landis, são uma das estruturas de dados fundamentais para a área. Elas pertencem à categoria das árvores de busca binária balanceadas e desempenham um papel crucial em algoritmos de pesquisa e operações de inserção/exclusão eficientes. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

Estudante, uma árvore binária pode ser percorrida de diversas formas. No percurso pré-ordem, primeiramente, visitamos a raiz da árvore e, depois, tentamos visitar a subárvore esquerda para, só então, visitar a subárvore direita. No percurso em ordem, primeiro a subárvore esquerda é visitada por completo para, só então, a raiz da árvore ser visitada e, ao fim, a subárvore direita é visitada.

O último caminhamento que vimos, o pós-ordem, faz a visitação da subárvore esquerda, depois da subárvore direita para, finalmente, visitar a raiz da árvore.

Também visualizamos como é possível estabelecer regras para criar árvores binárias ordenadas e balanceadas, para que seja possível realizar buscas de maneira otimizada. Para criar uma árvore binária de busca, podemos estabelecer que elementos menores ou iguais à raiz só podem ser inseridos na subárvore esquerda, e elementos maiores na subárvore direita. A partir disso, podemos realizar buscas com a mesma lógica.

Além disso, as árvores AVL nos dão ferramentas para rotacionar nós desbalanceados em busca de uma árvore binária tão balanceada quanto possível. Isso otimiza ainda mais os tempos de busca nesse tipo de estrutura de dados.



PENSANDO JUNTOS

Um bom algoritmo é como uma faca afiada: ele faz o que deveria fazer com um mínimo de esforço aplicado. Usando o algoritmo errado para resolver um problema é como tentar cortar um bife com uma chave de fenda: você pode eventualmente obter um resultado digerível, mas você gastará consideravelmente mais esforço do que o necessário, e é improvável que o resultado seja esteticamente agradável (CORMEN *et al.*, (2012)).

NOVOS DESAFIOS

Estudante, as árvores binárias têm uma importância significativa na vida de profissionais que desejam trabalhar com linguagem de programação e estrutura de dados, e em várias disciplinas como ciência da computação, análise e desenvolvimento de sistemas, engenharia de software, sistemas de informação,

desenvolvimento de dados e análise de algoritmos, entre outros. Aqui, estão algumas razões pelas quais as árvores binárias são relevantes e suas perspectivas no mercado de trabalho:

1. Estrutura de Dados Fundamental:

As árvores binárias são uma estrutura de dados fundamental e são usadas em uma ampla gama de aplicações. Profissionais de programação e desenvolvimento de software frequentemente encontram situações em que precisam armazenar e manipular dados de maneira hierárquica e eficiente, e as árvores binárias são uma escolha comum para esse fim.

2. Algoritmos de Busca e Classificação:

As árvores binárias desempenham um papel crucial em algoritmos de busca e classificação, como a árvore de busca binária e as árvores AVL ou Rubro-Negras. Entender como essas estruturas funcionam é fundamental para otimizar a busca e a ordenação de dados, habilidades essenciais para qualquer desenvolvedor.

3. Bancos de Dados:

Em bancos de dados, as árvores binárias são usadas para representar índices e estruturas de acesso eficiente aos dados. Profissionais que trabalham com bancos de dados relacionais ou NoSQL podem se beneficiar do conhecimento de como essas estruturas funcionam.

4. Redes de Computadores:

Em redes, as árvores binárias podem ser usadas em roteadores e sistemas de encaminhamento para efetuar decisões de roteamento eficientes. Engenheiros de rede podem se deparar com essas estruturas ao projetar e otimizar redes.

5. Aprendizado de Máquina e Inteligência Artificial:

Em campos como aprendizado de máquina e IA, as árvores de decisão binárias são usadas para representar modelos de classificação. Ter conhecimento sobre essas estruturas pode ser valioso ao desenvolver e otimizar algoritmos de aprendizado de máquina.

6. Perspectivas no Mercado de Trabalho:

Profissionais que dominam árvores binárias e suas aplicações têm uma vantagem competitiva no mercado de trabalho. Muitas empresas de tecnologia, startups e empresas de desenvolvimento de software procuram candidatos que tenham um sólido entendimento de estruturas de dados e algoritmos, incluindo árvores binárias.

7. Evolução Tecnológica:

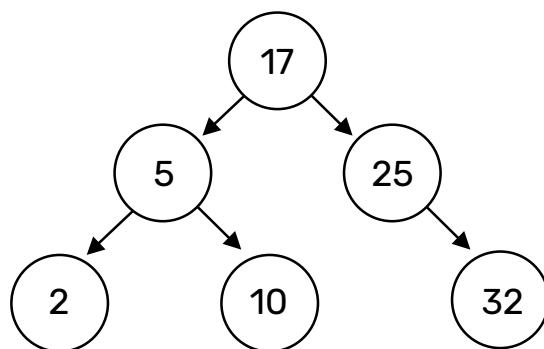
À medida que a tecnologia avança, a necessidade de eficiência e escalabilidade nos sistemas de software e hardware também aumenta. Isso torna as árvores binárias e outras estruturas de dados eficientes ainda mais relevantes, pois elas desempenham um papel crucial na otimização de recursos computacionais.

Em resumo, as árvores binárias são um conceito fundamental e versátil para o futuro profissional. Ter um bom entendimento dessas estruturas pode abrir portas no mercado de trabalho e ajudar os profissionais a enfrentar desafios técnicos complexos de maneira eficaz. Portanto, investir tempo em aprender e aprimorar suas habilidades em árvores binárias é uma escolha valiosa para qualquer pessoa que trabalhe ou pretenda trabalhar na área de tecnologia.

VAMOS PRATICAR

1. Os tipos de caminhamento em árvores binárias de busca são estratégias fundamentais para a análise e manipulação eficiente de dados armazenados nessa estrutura de dados. Um dos tipos de caminhamento mais conhecidos é o caminhamento em ordem, no qual os nós são visitados em ordem crescente dos valores dos elementos armazenados. Outra estratégia é o caminhamento pré-ordem que visita primeiro a raiz, seguida pelo nó esquerdo e, por fim, o nó direito - forma pré-fixada. O caminhamento pós-ordem, por sua vez, começa pelos nós filhos antes de visitar a raiz.

Considere a árvore binária a seguir:



- a) Qual a ordem de visitação dos nós ao se aplicar o percurso pré-ordem na árvore binária apresentada na figura?
b) Qual a ordem de visitação dos nós ao se aplicar o percurso em ordem na árvore binária apresentada na figura?
c) Qual a ordem de visitação dos nós ao se aplicar o percurso pós-ordem na árvore binária apresentada na figura?
2. As árvores binárias de busca são uma estrutura de dados utilizada para armazenar e gerenciar dados de maneira eficiente, permitindo buscas, inserções e exclusões em tempo logarítmico. Essas árvores possuem características especiais que as tornam muito úteis em uma variedade de aplicações, desde bancos de dados até algoritmos de ordenação.

VAMOS PRATICAR

Considere o seguinte conjunto de dados: 13, 70, 60, 39, 66, 55, 50, 42. Levando em conta a ordem dos elementos recém apresentados, monte uma árvore binária de busca e a partir da árvore de busca montada, realize uma busca pelo elemento 42 e informe quantos nós foram visitados para se encontrar tal elemento, considerando que a busca parte sempre da raiz da árvore.

3. As árvores binárias de busca são uma forma de estrutura de dados que organiza seus elementos de maneira hierárquica, onde cada nó possui no máximo dois filhos: um filho esquerdo e um filho direito. Os tipos de caminhamento em árvores binárias de busca são estratégias fundamentais para a análise e manipulação eficiente de dados armazenados nessa estrutura de dados.

Qual dos seguintes tipos de caminhamentos em árvores binárias explora os nós na seguinte ordem: nó esquerdo, nó direito, raiz?

- a) Pré-ordem.
- b) Pós-ordem.
- c) Em ordem.
- d) Nível-ordem.
- e) Simétrico-ordem.

4. Uma das técnicas de caminhamento em árvores binárias de busca envolve visitar a raiz primeiro, seguida pelos nós filhos. Especificamente, a sequência de passos deste caminhamento é a seguinte:

- I - Visitar o nó raiz.
- II - Realizar o caminhamento no nó filho esquerdo.
- III - Realizar o caminhamento no nó filho direito.

Esse tipo de caminhamento tem diversas aplicações em algoritmos e estruturas de dados. Ele é especialmente útil quando se deseja realizar operações que requerem uma abordagem pré-fixada na árvore.

VAMOS PRATICAR

Qual dos seguintes tipos de caminhamentos em árvores binárias explora os nós na seguinte ordem: raiz, filho esquerdo, filho direito?

- a) Pré-ordem.
 - b) Pós-ordem.
 - c) Em ordem.
 - d) Nível-ordem.
 - e) Simétrico-ordem.
5. Para resolver o problema do desbalanceamento de árvores binárias de busca, os pesquisadores Adelson-Velskii e Landis, em 1962, criaram um algoritmo que leva as iniciais de seus nomes. As árvores AVL são, nesse sentido, árvores nas quais todos os nós encontram-se平衡ados.

Com isso em mente, analise as afirmativas a seguir:

- I - Árvores AVL nunca precisam ser reequilibradas, pois mantêm automaticamente seu balanceamento.
- II - As operações de inserção e remoção em árvores AVL podem levar a uma situação de desbalanceamento.
- III - Para tornar a árvore balanceada, precisamos realizar rotações.
- IV - Em árvores AVL, a rotação é um processo que sempre resulta em um desbalanceamento maior.

É correto o que se afirma em:

- a) I e IV, apenas.
- b) II e III, apenas.
- c) III e IV, apenas.
- d) I, II e III, apenas.
- e) II, III e IV, apenas.

REFERÊNCIAS

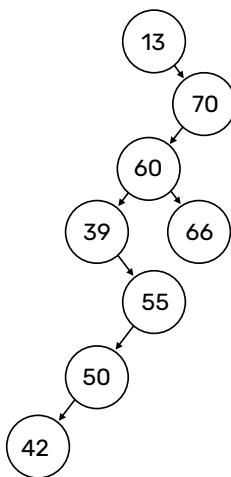
- ADELSON-VELSKII, G.; LANDIS, E. M. An algorithm for the organization of information - **Proceedings of the USSR Academy of Sciences**, v. 146, p. 263-266, 1962.
- CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN C. **Algoritmos: Teoria e Prática**. Rio de Janeiro: Elsevier, 2012.
- DROZDEK, A. **Estrutura de dados e algoritmos em C++**. 2. ed. São Paulo: Cengage Learning, 2016.
- OLIVEIRA, P. M .de; PEREIRA, R. de L. **Estrutura de Dados II**. Maringá: Unicesumar, 2019. 152 p.

GABARITO

1.

- a) $17 - 5 - 2 - 10 - 25 - 32$.
- b) $2 - 5 - 10 - 17 - 25 - 32$.
- c) $2 - 10 - 5 - 32 - 25 - 17$.

2.



A ordem de visitação para chegar até o 42 é a seguinte: 13, 70, 60, 39, 55, 50, 42, ou seja, sete nós foram visitados.

- 3. Opção B.** No caminhamento pós-ordem, também conhecido por caminhamento pós-fixado, primeiramente, visitamos toda a subárvore esquerda, depois, toda a subárvore direita. Só após ter visitado as duas subárvore, é que marcamos o nó corrente como visitado.
- 4. Opção A.** O caminhamento pré-ordem, também conhecido por caminhamento pré-fixado, marca, primeiramente, a raiz como visitada, e só depois visitamos as subárvore esquerda e direita, respectivamente.
- 5. Opção B.** Uma árvore balanceada pode perder essa característica quando um novo elemento é inserido. Isso também pode ocorrer quando um elemento é removido da árvore binária. Assim, quando ocorrem operações de inserção ou remoção em árvores AVL, recalculam-se os fatores de平衡amento de cada nó para, assim, poder executar rotações nos nós problemáticos, na tentativa de restabelecer seu balanceamento.



TEMA DE APRENDIZAGEM 8

GRAFOS

MINHAS METAS

- Conhecer a origem dos grafos.
- Aprender suas propriedades.
- Representar grafos computacionalmente.
- Estudar exemplos de aplicação de grafo na modelagem de problemas.
- Implementar grafos em linguagem C.

INICIE SUA JORNADA

Estudante, veremos um dos assuntos mais pesquisados e interessantes dentro da Ciência da Computação, a **Teoria dos Grafos**. Mais do que uma estrutura de dados, os grafos permitem modelar, de forma matemática, problemas reais de logística, custos, eficiência, dentre muitos outros.

Um exemplo da aplicação de grafo para a solução de problemas é demonstrado por Tanenbaum (2003, p. 375-377) no capítulo sobre **Roteamento pelo caminho mais curto**. O grafo representa os roteadores e as rotas entre eles, e, com o algoritmo de Dijkstra, é possível encontrar a menor rota entre dois roteadores na rede.

Sabe quando você busca um endereço pelo Google Maps ou num GPS? Aquelas informações das ruas estão armazenadas em forma de grafo e algoritmos conhecidos e estão aplicados para apresentar possíveis caminhos de onde você está no momento até o destino desejado. Agora faça o teste você mesmo! Essa atividade prática e interativa permite que você veja o mundo real por trás das tecnologias que usamos todos os dias.

Neste tema, vamos aprender onde o grafo surgiu e quem o criou. Abordaremos a parte teórica da sua definição, seu conceito e a sua utilização. Ensinaremos, também, a forma de criar uma representação gráfica a partir de um conjunto de dados, e vice-versa. Finalmente, ensinaremos uma das muitas formas de modelagem computacional de grafos.



PLAY NO CONHECIMENTO

Os grafos são uma ferramenta poderosa e versátil na ciência da computação e em muitas outras áreas, permitindo-nos analisar e compreender a complexidade de sistemas interconectados. Que tal desvendar suas aplicações práticas por meio de exemplos concretos? Então, dê play no podcast: *Desvendando Conexões: Grafos e Suas Aplicações Práticas no Mundo Real!* **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

DESENVOLVA SEU POTENCIAL

GRAFOS

Sete pontes de Königsberg

A **Teoria dos Grafos** surgiu, informalmente, em 1736, quando o matemático e físico suíço Leonhard Paul Euler (1707-1783), por meio do seu artigo *Solutio problematis ad geometriam situs pertinentes*, propôs uma solução para o famoso problema matemático conhecido como **Sete pontes de Königsberg**.

A região de Königsberg (atual Kaliningrado) era (e ainda é) cortada pelo Rio Prególia, que divide o território em duas grandes ilhas e duas faixas continentais. Na época, havia 7 pontes que interligavam todo o complexo geográfico que formava a cidade.

Bridges of Königsberg

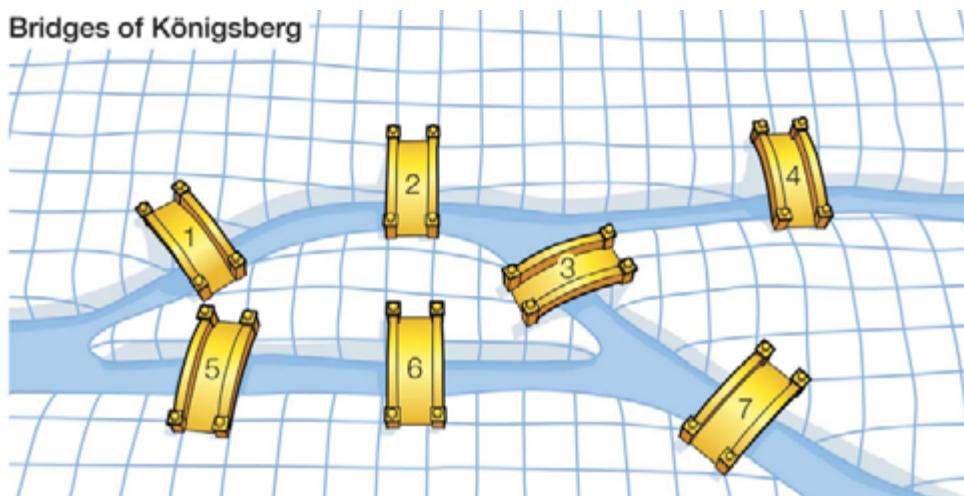


Figura 1 - Sete pontes de Königsberg / Fonte: Oliveira, Pereira (2019, p. 94).

Descrição da Imagem: a figura demonstra duas faixas continentais separadas por um rio e duas ilhas no meio desse rio. Sete pontes conectam os dois lados do continente com as ilhas. As pontes um e dois ligam o continente de cima com a primeira ilha. A ponte três conecta a primeira ilha com a segunda. A ponte quatro conecta o continente de cima com a segunda ilha. As pontes cinco e seis conectam a primeira ilha com o continente de baixo e, finalmente, a ponte sete conecta a segunda ilha com o continente de baixo. Fim da descrição.

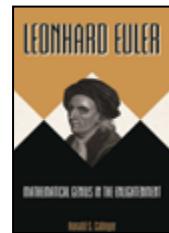


INDICAÇÃO DE LIVRO

Leonhard Euler: Mathematical Genius in the Enlightenment

Autor: Ronald S. Calinger

Sobre o Livro: um livro muito recomendado para aprender mais sobre a vida e o trabalho de Leonhard Euler, um dos matemáticos mais influentes da história. Trata-se de uma biografia abrangente, que mergulha na vida e nas realizações de Euler, revelando suas contribuições significativas para a matemática, física e engenharia.



Em uma época sem internet, TV a cabo e telefones celulares, a população possuía poucas opções de lazer. Uma delas era solucionar uma lenda popular que dizia ser possível atravessar todas as sete pontes de Königsberg sem repetir nenhuma delas no trajeto.

Leonhard Euler provou matematicamente que não havia uma solução possível de satisfazer tais restrições. Para isso, ele modelou o problema de forma abstrata. Considerou cada porção de terra como um ponto (vértice) e cada ponte como uma reta (aresta) que ligava dois pontos (duas porções de terra). Esse é o registro científico formal mais antigo de um grafo.

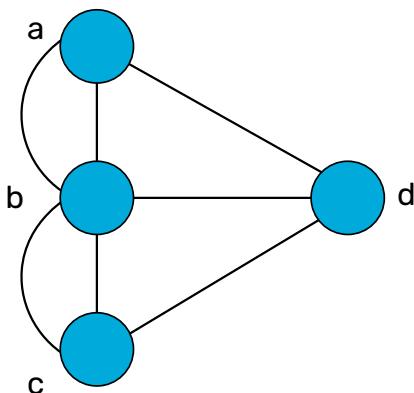


Figura 2 - Grafo criado por Euler para representar o problema das sete pontes de Königsberg
Fonte: Oliveira, Pereira (2019, p. 94).

Descrição da Imagem: a figura representa um grafo onde existem três vértices um abaixo do outro de letras a, b e c e um quarto vértice ao lado do vértice b, de letra d. Linhas conectam os vértices formando dois triângulos retângulos, de vértices, a, b e d e b, c e d. Duas linhas ovaladas ligam o vértice a com b e b com c. Fim da descrição.

Analizando a Figura 2, ele percebeu que só seria possível caminhar todo o percurso atravessando cada uma das pontes uma única vez se houvesse exatamente zero ou dois vértices de onde saísse um número ímpar de arestas. Falar é fácil, o difícil é provar que isso é uma verdade. Vamos olhar atentamente a figura criada por ele e tentar chegar à mesma conclusão.

Eu posso passar quantas vezes eu quiser por cada uma das porções de terra (pontos), o que eu não posso é repetir as pontes (retas). Se um vértice possuir um número par de arestas, eu posso chegar àquele ponto pela primeira aresta e sair pela segunda. Não havendo vértices com número ímpar de arestas, o percurso pode ser iniciado a partir de qualquer vértice, porém deverá iniciar e terminar no mesmo ponto no grafo (Figura 3). Isso prova que a primeira afirmação feita por Euler é verdadeira: deve haver exatamente **zero vértice** com número ímpar de arestas.

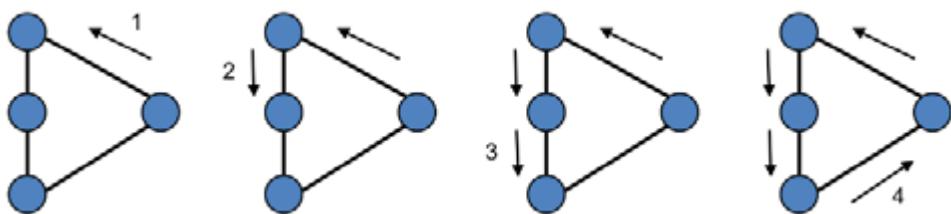


Figura 3 - Percorrendo um grafo apenas com vértices de número par de arestas.
Fonte: Oliveira, Pereira (2019, p. 95).

Descrição da Imagem: a figura contém 4 repetições de 3 círculos representando vértices, três deles um abaixo do outro de letras a, b e c e um quarto ao lado da de letra b de letra d. uma linha liga os vértices formando um triângulo isósceles. Na primeira figura uma seta de número 1 aponta do vértice d para o a. Na segunda figura a seta 1 se mantém e uma seta de número 2 aponta de a para b. Na terceira figura é acrescentada uma seta de b para c e na quarta figura é acrescentada uma seta de número 4 de c para d. Fim da descrição.

No entanto, para a prova final, precisamos testar a veracidade da segunda afirmação feita por ele. No caso de haver vértices com número ímpar de arestas, precisa haver exatamente dois deles. Se um ponto possuir um número ímpar de arestas, eu posso entrar e sair usando duas delas, sobrando uma terceira. Quando temos exatamente dois pontos com números ímpares de caminhos, um desses vértices será obrigatoriamente o início e outro o final do trajeto (Figura 4).

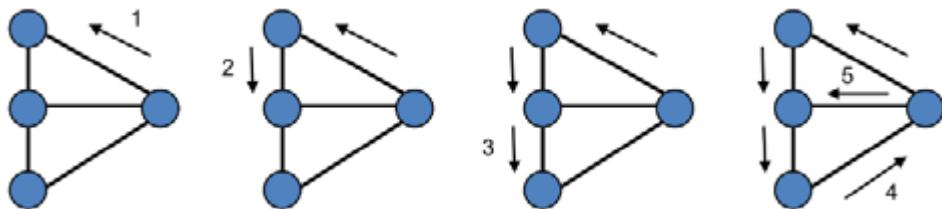


Figura 4 - Grafo com exatamente dois vértices com número ímpar de arestas
Fonte: Oliveira, Pereira (2019, p. 95).

Descrição da Imagem: a figura contém quatro repetições de três círculos representando vértices, três deles um abaixo do outro de letras a, b e c e um quarto ao lado da de letra b de letra d. uma linha liga os vértices formando dois triângulos retângulos. Na primeira figura, uma seta de número um aponta do vértice d para o a. Na segunda figura a seta um se mantém e uma seta de número dois aponta de a para b. Na terceira figura, é acrescentada uma seta de b para c e na quarta figura é acrescentada uma seta de número quatro de c para d e uma seta de número cinco de d para b. Fim da descrição.

Teoria dos grafos

A **teoria dos grafos** é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. Para isso, são empregadas estruturas chamadas de grafos.

Grafo é uma estrutura $G = (V, E)$, em que V é **um conjunto finito não nulo** de vértices (ou nós), e E é **um conjunto de arestas (ou arcos)**. Uma aresta é **um** par de vértices $a = \{v, w\}$, em que v e $w \in V$ e $a \in E$.

Após a apresentação formal entre você, estudante, e o dito grafo, vamos analisar novamente o desenho criado por Euler (ver Figura 2) e traduzir isso para uma linguagem menos matemática.

Como acabamos de apresentar, o grafo é formado por dois conjuntos: um de vértices (V , que não pode ser nulo) e um de arestas (E). Dado o grafo de Euler, podemos dizer que o conjunto V é formado por:

$$V = \{a, b, c, d\}$$

As arestas são formadas sempre por dois vértices que existam no conjunto V e não há nenhuma obrigatoriedade da existência de arestas no grafo. Com isso, em mente, concluímos que E tem a seguinte configuração:

$$E = \{(a, b), (b, c), (c, d), (d, a)\}$$

No caso de os pares de vértices serem ordenados, ou seja, uma aresta $a = (v, w)$ é considerada diferente da aresta $a = (w, v)$, o grafo é dito **orientado** (ou **dígrafo**) (Figura 5).

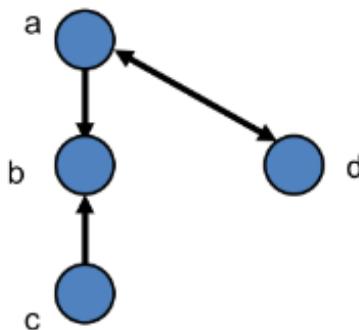


Figura 5 – Exemplo de grafo orientado / Fonte: Oliveira, Pereira (2019, p. 97).

Descrição da Imagem: a figura representa um grafo onde existem três vértices um abaixo do outro de letras a, b e c e um quarto vértice ao lado do vértice b, de letra d. Uma seta de ponta dupla liga o vértice a e d. Uma seta aponta do vértice a para o b e outra do vértice c para o b. Fim da descrição.

Para indicar que os elementos do conjunto E são pares ordenados, usamos a notação de chaves angulares, em vez de parênteses. A representação do conjunto E do grafo anterior é a seguinte:

$$E = \{(a, b), (c, b), (a, d), (d, a)\}$$

Em um grafo simples, dois vértices v e w são **adjacentes** (ou vizinhos) se há uma aresta $a = \{v, w\}$ em G . Esta aresta é dita ser **incidente** a ambos, v e w . No caso de grafos orientados, diz-se que cada aresta $a = (v, w)$ possui uma única direção de v para w , onde a aresta $a = (v, w)$ é dita **divergente** de v e **convergente** a w .

O grau de um vértice é definido pela sua quantidade de arestas. O seu grau de saída é a quantidade de arestas divergentes e o grau de entrada o de arestas convergentes. No exemplo de grafo orientado, podemos afirmar que o vértice *a* possui grau 3, sendo grau de saída 2 {(a, b), (a, d)} e grau 1 de entrada {(d, a)}.

Um grafo é dito grafo conexo quando é possível partir de um vértice *v* até um vértice *w*, por meio de suas arestas incidentes. Caso contrário, o grafo é dito **desconexo** (Figura 6).

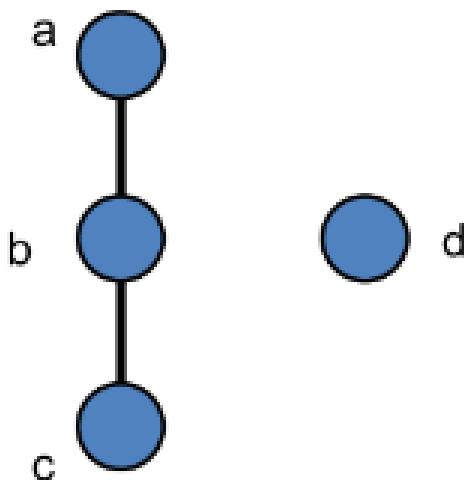


Figura 6 - Exemplo de grafo desconexo / Fonte: Oliveira, Pereira (2019, p. 97).

Descrição da Imagem: a figura representa um grafo onde existem três vértices um abaixo do outro de letras a, b e c e um quarto vértice ao lado do vértice b, de letra d. Uma linha conecta o vértice a com o b e b com o c. Fim da descrição. Fim da descrição.

Vimos, até aqui, que é possível desenhar um grafo a partir de um conhecido conjunto $G = (V, E)$ da mesma forma que, a partir de uma representação gráfica, podemos encontrar os elementos pertencentes aos conjuntos V e E .

Grafos como representação de problemas

O grafo é uma estrutura muito interessante e versátil. Ela permite modelar de forma matemática diversos problemas reais existentes no nosso cotidiano. Foi o que Euler fez em 1736. A partir do modelo matemático, é possível procurar a solução por meio de inúmeros algoritmos já criados.

Por exemplo, no desenvolvimento de um site, você pode considerar cada página como um vértice e o link entre duas delas como uma aresta. Assim, você pode representar toda a organização e fluxo do site por meio de um grafo. Isso pode ajudar na hora de verificar se todas as páginas estão ligadas, se o usuário tem a possibilidade de acessar todo o conteúdo e se a estrutura tem boa naveabilidade.

Uma empresa de logística pode considerar cada cliente como um nó e o caminho entre dois deles um arco. Por meio de um algoritmo de busca de caminho, como o algoritmo de busca em largura (BFS-Breadth First Search), um sistema informatizado poderia traçar a rota de entrega para o dia seguinte de forma automática a partir de uma lista de pedidos.

Ainda usando o exemplo da empresa de logística, imagine que cada aresta do gráfico possui um peso que indica a distância entre dois locais de entrega. Existem algoritmos como o de Dijkstra (um BFS guloso), capaz de não apenas buscar o caminho entre dois pontos, mas procurar o caminho de menor custo, mais eficiente ou de maior lucro.

Podemos aplicar isso em outros problemas, como numa empresa de transporte aéreo. Imagine que cada aeroporto/cidade seja considerado um vértice no gráfico, a rota (voo) entre eles uma aresta. Se um passageiro decidir viajar de Curitiba no Paraná até Manaus no Amazonas e não há um voo direto entre essas duas capitais, o sistema pode buscar e oferecer várias opções de conexões entre diferentes rotas para o viajante.

Os pesos nas arestas não precisam necessariamente representar o custo monetário entre dois nós. Podemos considerar outras grandezas com o tempo de espera no aeroporto, o tempo de viagem, modelo da aeronave e assim por diante.

Representação computacional de grafos

Existem inúmeras formas de representar computacionalmente um grafo, cada qual com suas vantagens e desvantagens em relação a tempo de implementação, uso de memória, gasto de processamento etc. Por questões pedagógicas, vamos apresentar a mais prática.

Sabemos que um grafo é uma estrutura $G = (V, E)$, em que V é um conjunto finito não nulo de vértices ou nós.

VOCÊ SABE RESPONDER?

Qual a maneira mais simples e prática para representar um conjunto finito de informações?

Acertou quem se lembrou do nosso velho conhecido **vetor**.

Ainda pensando na estrutura do grafo G , E é um conjunto de arestas ou arcos. Uma aresta é um par de vértices $a = \{v, w\}$, em que v e $w \in V$ e $a \in E$. Para a representação de pares ordenados, podemos utilizar uma **matriz bidimensional**.

Dada uma matriz M_{ij} , em que i é a quantidade de linhas e j a quantidade de colunas, podemos dizer que, se houver uma aresta a ligando os vértices v e w , então o valor contido em M_{vw} será 1. Essa matriz é chamada de **Matriz de Adjacência**, lembrando que dois vértices são adjacentes num grafo se houver uma aresta entre eles.

Vamos ilustrar essas definições usando o seguinte grafo:

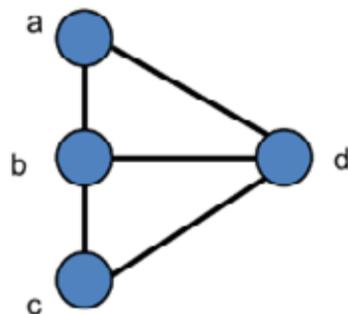


Figura 7 - Representação de um grafo / Fonte: Oliveira, Pereira (2019, p. 99).

Descrição da Imagem: a figura representa um grafo onde existem três vértices um abaixo do outro de letras a, b e c e um quarto vértice ao lado do vértice b, de letra d. Linhas conectam os vértices formando dois triângulos retângulos, de vértices, a, b e d e b, c e d. Fim da descrição.

O vetor que representa os seus vértices é representado por:

$$V = \{a, b, c, d\}$$

Como o grafo possui quatro vértices, a sua matriz de adjacência terá quatro linhas e quatro colunas. Cada intersecção entre linha e coluna representa um par ordenado (aresta) no grafo.

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 0 & 0 & 0 \\ b & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 \\ d & 0 & 0 & 0 & 0 \end{bmatrix}$$

Sabemos que existe uma aresta ligando os vértices a e b , então, o valor de M_{ab} .

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 \\ d & 0 & 0 & 0 & 0 \end{bmatrix}$$

Como se trata de um grafo não ordenado, podemos dizer que de $M_{ab} = M_{ba}$. Dessa forma, é verdade dizer que M_{ba} também é 1.

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 1 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 \\ d & 0 & 0 & 0 & 0 \end{bmatrix}$$

Usando desse raciocínio, vamos completar a matriz M alterando para 1 o valor de cada par ordenado que representar uma aresta entre dois vértices do grafo.

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & 0 & 1 \\ b & 1 & 0 & 1 & 1 \\ c & 0 & 1 & 0 & 1 \\ d & 1 & 1 & 1 & 0 \end{bmatrix}$$

A representação de um conjunto E de vértices na forma de uma Matriz de Adjacências também facilita a visualização de outros tipos de informação.

Por exemplo, sabemos que o vértice a é de grau dois, pois possui duas arestas $\{(a, b), (a, d)\}$. Numa matriz de adjacência o grau de um determinado vértice é dado pela quantidade de números 1 na sua linha ou coluna.

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & 0 & 1 \\ b & 1 & 0 & 1 & 1 \\ c & 0 & 1 & 0 & 1 \\ d & 1 & 1 & 1 & 0 \end{bmatrix}$$

Vamos criar agora uma matriz de adjacências para um grafo orientado.

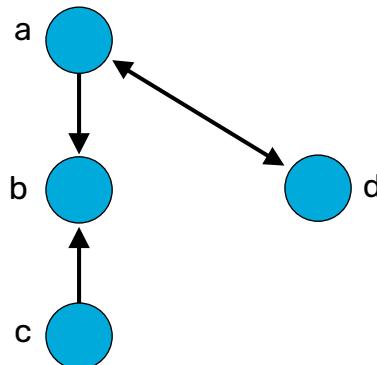


Figura 8 - Representação de um grafo orientado / Fonte: Oliveira, Pereira (2019, p. 101).

Descrição da Imagem: a figura representa um grafo onde existem três vértices um abaixo do outro de letras a, b e c e um quarto vértice ao lado do vértice b, de letra d. Uma seta de ponta dupla liga o vértice a e d. Uma seta aponta do vértice a para o b e outra do vértice c para o b. Fim da descrição.



PENSANDO JUNTOS

Quando se trata de um grafo não orientado, podemos dizer que um vértice $a = \{v, w\}$ é igual ao vértice $a = \{w, v\}$, porém em um grafo orientado isso não é verdade.

É possível dizer o grau de saída de um vértice contando a quantidade de números 1 na sua linha, e o seu grau de entrada pela quantidade de números 1 na sua coluna.

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 0 \\ c & 0 & 1 & 0 & 0 \\ d & 1 & 0 & 0 & 0 \end{bmatrix}$$

Nesse exemplo, o grau do vértice a é 3, sendo grau 2 de saída e grau 1 de entrada.

Implementando grafos em C

Agora que você já viu a Teoria dos Grafos e aprendeu uma forma simples de representá-los de forma computacional, vamos colocar a mão na massa. **Não há maneira melhor de aprender a programar do que programando.**

Não iremos trabalhar aqui com estruturas dinâmicas, então, devemos utilizar de estruturas estáticas para armazenar as informações do grafo.

O grafo é composto por dois elementos, sendo o primeiro um conjunto de vértices e o segundo um conjunto de arestas.

Para representar os vértices, vamos usar um vetor; para as arestas, uma matriz (vetor bidimensional).

Como não sabemos a quantidade de vértices no grafo, vamos criar um vetor suficientemente grande para que acomode uma grande quantidade de informações. Para isso, vamos definir uma constante chamada *maximo* de valor 10.

```
//Constantes
#define maximo 10
```

Quadro 1 - Criação de vetor - parte 1 / Fonte: Oliveira, Pereira (2019, p. 102).

Como variáveis, vamos criar um vetor chamado *grafo*, que terá o tamanho definido na constante *maximo*. Para as arestas, usaremos um vetor bidimensional chamado *ma*, que representará uma matriz de adjacências. Vamos permitir no nosso projeto que o usuário defina o tamanho do grafo até o limite estabelecido em *maximo*. Uma variável chamada *tamanho* do tipo inteira é o suficiente para tal tarefa.

```
//Constantes
#define maximo 10
//Variáveis
int tamanho=0;
int grafo[maximo];
int ma[maximo][maximo];
```

Quadro 2 - Criação de vetor - parte 2 / Fonte: Oliveira, Pereira (2019, p. 103).

O vetor foi criado com um tamanho muito grande, mas o usuário poderá usar apenas uma parte dele. Vamos criar uma função chamada *grafo_tamanho()*, que lê a quantidade de vértices e retorna essa informação para a chamada da função.

```
//Define o número de vértices do Grafo
int grafo_tamanho(){
    int tamanho;
    printf("Escolha a quantidade de vértices do grafo: ");
    scanf("%d", &tamanho);
    return tamanho;
}
```

Quadro 3 - Criação de vetor - parte 3 / Fonte: Oliveira, Pereira (2019, p. 102).

Como o foco aqui é demonstrar como criar a estrutura de um grafo em linguagem C, vamos nos concentrar apenas nas funções principais.

Se você vai deixar o usuário escolher a quantidade de nós, precisa verificar se esse valor é válido, ou seja, se é maior do que zero e menor ou igual ao tamanho definido no vetor.

Agora que eu já sei quantos vértices tem no grafo, o próximo passo é inserir as arestas. Numa matriz de adjacências, a representação de uma aresta entre um vértice x qualquer com um vértice y se dá pelo valor 1 na posição xy da matriz.

Como não se trata de um grafo orientado, podemos considerar que se existe uma aresta xy , também é verdadeiro considerar que existe uma aresta yx . Assim, a função `grafo_inserir()` perguntará para o usuário os dois vértices e irá adicionar o valor 1 nas respectivas posições dentro da matriz de adjacências.

```
//Inserir aresta
void grafo_inserir(){
    int num1, num2;
    system("cls");
    printf("Escolha o vértice de origem entre 0 a %d: ",tamanho-1);
    scanf("%d",&num1);
    printf("Escolha o vértice de destino entre 0 a %d: ",tamanho-1);
    scanf("%d",&num2);
    if (num1 > tamanho-1 || num2 > tamanho-1 || num1 <0 || num2 <0) {
        printf("\nOs valores precisam estar entre 0 e %d\n\n",tamanho);
        system("pause");
    }
    else {
        ma [num1] [num2]=1;
        ma [num2] [num1]=1;
    }
}
```

Quadro 4 - Criação de vetor - parte 4 / Fonte: Oliveira, Pereira (2019, p. 104).

Precisamos, também, do efeito oposto: remover uma aresta no grafo. Não queremos que o usuário comece tudo do zero, caso tenha inserido uma aresta por engano. A função *grafo_remover()* também será útil em momentos em que seja necessário transformar o grafo a partir da remoção de arestas. O procedimento será o mesmo, só que, em vez de colocar o valor 1 na posição *xy* da matriz de adjacências, o valor será definido como 0.

```
//Remover aresta
void grafo_remover() {
    int num1, num2;
    system("cls");
    printf("Escolha o vértice de origem entre 0 a %d: ",tamanho);
    scanf("%d", &num1);
    printf("Escolha o vértice de destino entre 0 a %d: ",tamanho);
    scanf("%d", &num2);
    if (num1 > tamanho-1 || num2 > tamanho-1 || num1 <0 || num2 <0) {
        printf("\nOs valores precisam estar entre 0 e %d\n\n",tamanho);
        system("pause");
    }
    else {
        ma [num1] [num2]=0;
        ma [num2] [num1]=0;
    }
}
```

Quadro 5 - Criação de vetor - parte 5 / Fonte: Oliveira, Pereira (2019, p. 104).

Com os dados na memória, só falta mostrar o resultado para o usuário. Vamos criar duas funções, uma chamada *grafo_desenhar()*, que apresentará na tela a lista de vértices, e outra chamada *grafo_desenhar_ma()*, para desenhar a matriz de adjacências.

Para o vetor de vértices é muito simples, basta um laço de repetição que comece em zero e vá até o valor contido na variável *tamanho*. Para cada interação no laço, o programa deve imprimir na tela o valor da posição no vetor.

```
//Função para desenhar o vetor de vértices
void grafo_desenhar(){
    //Desenhando lista de vértices
    printf("Listas de vértices\n[ ");
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", grafo[i]);
    }
    printf("]\n\n");
}
```

Quadro 6 - Criação de vetor - parte 6 / Fonte: Oliveira, Pereira (2019, p. 105).

Já para desenhar a matriz, precisaremos de dois laços de repetição, um percorrendo todas as linhas e, para cada linha, o segundo laço percorre todas as colunas.

```
//Função para desenhar a matriz de arestas
void grafo_desenhar_ma(){
    //Desenhando matriz de adjacências
    printf("Matriz de adjacencias\n[\\n");
    for (int i = 0; i < tamanho; i++) {
        for (int j = 0; j < tamanho; j++) {
            printf(" %d", ma[i][j]);
        }
        printf("\\n");
    }
    printf("]\n\n");
}
```

Quadro 7 - Criação de vetor - parte 7 / Fonte: Oliveira, Pereira (2019, p. 105).

A seguir, você encontrará o código completo da implementação de um Grafo em linguagem C. Digite o código no seu compilador e faça alguns testes incluindo e removendo arestas. Experimente recriar no seu programa os grafos vistos nesta unidade.

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

```
//Constantes
#define maximo 10

//Variáveis
int tamanho=0;
int grafo[maximo];
int ma[maximo][maximo];
int op=1;

//Prototipação
int grafo_tamanho();
void grafo_desenhar();
void grafo_desenhar_ma();
void grafo_inserir();
void grafo_remover();
void menu_mostrar();

//Função Principal
int main(){
    while (tamanho <= 0 || tamanho > maximo) {
        tamanho = grafo_tamanho();
        if(tamanho <= 0 || tamanho > maximo) {
            system("cls");
            printf("Escolha um valor entre 1 e %d!\n\n", maximo);
        }
        else {
            for(int i=0; i<tamanho;i++){
                grafo[i]=i;
            }
        }
    }
    while (op != 0) {
        system("cls");
        grafo_desenhar();
        grafo_desenhar_ma();
        menu_mostrar();
        scanf("%d", &op);
        switch (op) {
            case 1:
                grafo_inserir();
                break;
            case 2:
                grafo_remover();
                break;
        }
    }
}
```

```
}

system("Pause");
return(0);
}

//Define o número de vértices do Grafo
int grafo_tamanho(){
    int tamanho;
    printf("Escolha a quantidade de vértices do grafo: ");
    scanf("%d", &tamanho);
    return tamanho;
}

//Função para desenhar o vetor de vértices
void grafo_desenhar(){
    //Desenhando lista de vértices
    printf("Listas de vértices\n[ ");
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", grafo[i]);
    }
    printf("]\n\n");
}

//Função para desenhar a matriz de arestas
void grafo_desenhar_ma(){
    //Desenhando matriz de adjacências
    printf("Matriz de adjacencias\n[\\n");
    for (int i = 0; i < tamanho; i++) {
        for (int j = 0; j < tamanho; j++) {
            printf(" %d", ma[i][j]);
        }
        printf("\\n");
    }
    printf("]\n\n");
}

//Inserir aresta
void grafo_inserir(){
    int num1, num2;
    system("cls");
    printf("Escolha o vértice de origem entre 0 a %d: ",tamanho-1);
    scanf("%d",&num1);
    printf("Escolha o vértice de destino entre 0 a %d: ",tamanho-1);
```

```

scanf ("%d", &num2);
if (num1 > tamanho-1 || num2 > tamanho-1 || num1 <0 || num2
<0) {
    printf ("\nOs valores precisam estar entre 0 e %d\n\n", tama-
    nho);
    system("pause");
}
else {
    ma [num1] [num2]=1;
    ma [num2] [num1]=1;
}
}

//Remover aresta
void grafo_remover () {
    int num1, num2;
    system("cls");
    printf ("Escolha o vértice de origem entre 0 a %d: ", tamanho);
    scanf ("%d", &num1);
    printf ("Escolha o vértice de destino entre 0 a %d: ", tamanho);
    scanf ("%d", &num2);
    if (num1 > tamanho-1 || num2 > tamanho-1 || num1 <0 || num2
    <0) {
        printf ("\nOs valores precisam estar entre 0 e %d\n\n", tama-
        nho);
        system("pause");
    }
    else {
        ma [num1] [num2]=0;
        ma [num2] [num1]=0;
    }
}

//Mostrar o menu de opções
void menu_mostrar () {
    printf ("\nEscolha uma opção:\n");
    printf ("1 - Inserir aresta\n");
    printf ("2 - Remover aresta\n");
    printf ("0 - Sair\n\n");
}

```

Quadro 8 - Criação de vetor - parte 8 / Fonte: Oliveira, Pereira (2019, p. 106-108).

**EU INDICO****Descubra o Segredo dos Caminhos Mais Curtos com o Algoritmo de Dijkstra!**

Você já se perguntou como os aplicativos de navegação determinam o caminho mais curto de um ponto A para um ponto B? Ou como as redes de computadores encontram o caminho mais eficiente para transmitir dados de um lugar para outro? A resposta está no Algoritmo de Dijkstra, uma ferramenta poderosa usada para calcular o caminho de custo mínimo em diversos cenários.

Acesse o site de Algoritmo de Dijkstra agora e comece a explorar. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

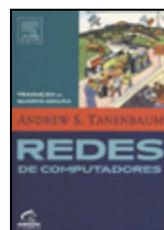
Os grafos são estruturas fascinantes e com eles, podemos representar de forma gráfica ou matemática diversos problemas reais presentes no nosso dia a dia. A gente não percebe, mas diversas facilidades existentes hoje na vida moderna são graças à aplicação de algoritmos especializados em resolver problemas baseados em grafos. Muitas empresas modelam, em grafo, a sua cadeia de produção, distribuição e logística, aplicando estudos e algoritmos para maximizar lucro, diminuir custo, otimizar o tempo.

Se um problema puder ser representado na forma de um grafo, é muito possível que exista um algoritmo que possa encontrar uma boa solução dentro de um conjunto desconhecido de possibilidades. Se esse algoritmo ainda não existir, você já tem conhecimento suficiente para criá-lo!

**INDICAÇÃO DE LIVRO****Redes de Computadores**

Autor: Andrew S. Tanenbaum

Sobre o Livro: se você está interessado em mergulhar no mundo das redes de computadores e deseja adquirir conhecimento sólido e prático sobre o assunto, este livro é a escolha perfeita. Não apenas irá equipá-lo com uma base sólida, mas também o manterá atualizado em um campo em constante evolução. Prepare-se para uma jornada fascinante no mundo das redes de computadores com este livro como seu guia confiável.





EM FOCO

Assista agora ao conteúdo referente a sua aula. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

NOVOS DESAFIOS

Estudante, à medida que o mundo se torna cada vez mais digital e interconectado, o conhecimento em grafos se torna uma habilidade essencial em várias áreas. Profissionais com expertise em grafos têm um leque vasto de oportunidades no mercado de trabalho. Por isso, vamos explorar onde e como você pode aplicar esse conhecimento de maneira impactante.

Tecnologia da Informação (TI):

Empresas de tecnologia, como gigantes da indústria de redes sociais, motores de busca e empresas de análise de dados, frequentemente, usam grafos para modelar conexões entre usuários, recomendações de produtos e análise de redes. Como um profissional em grafos, você pode desempenhar um papel crucial na otimização de algoritmos de busca, detecção de fraudes, personalização de conteúdo e muito mais.

Transporte e Logística:

Empresas de transporte, como aplicativos de mobilidade e empresas de logística, utilizam grafos para planejar rotas eficientes, rastrear entregas e otimizar a logística de armazenamento. Como especialista em grafos, você pode contribuir para a eficiência operacional dessas organizações.

Redes de Telecomunicações:

Na área de telecomunicações, grafos são usados para modelar a infraestrutura de rede, analisar o tráfego de dados e garantir a conectividade confiável. Profissionais em grafos são fundamentais para a expansão e a manutenção de redes de comunicação.

Ciências da Computação e Pesquisa Acadêmica:

A pesquisa em algoritmos e teoria de grafos é uma área acadêmica em crescimento constante. Se você tem um interesse profundo em resolver problemas complexos, uma carreira na pesquisa em grafos pode ser a escolha certa para você.

Setor Financeiro:

Bancos e instituições financeiras usam grafos para modelar transações, detecção de fraudes e análise de riscos. Especialistas em grafos desempenham um papel fundamental na segurança e eficiência do setor financeiro.

Bioinformática e Ciências da Vida:

Na pesquisa biomédica e genômica, os grafos são usados para representar interações entre proteínas, análise de dados genéticos e modelagem de redes metabólicas. Esse campo em constante evolução oferece oportunidades empolgantes para profissionais em grafos que desejam contribuir para a melhoria da saúde humana.

Gestão de Projetos e Logística Empresarial:

Empresas que lidam com grandes operações, como cadeias de suprimentos e gestão de projetos de construção, podem se beneficiar da aplicação de grafos para otimizar processos, planejamento de recursos e tomada de decisões estratégicas.

Como profissional em grafos, você poderá trabalhar em uma variedade de setores e contribuir para resolver problemas complexos de forma eficiente e inovadora. Sua capacidade de modelar e analisar relacionamentos e conexões será uma habilidade valiosa, independentemente de onde você escolher atuar.

Portanto, se você está interessado em uma carreira que oferece desafios estimulantes e um impacto significativo em diferentes indústrias, explorar o mundo dos grafos pode ser a chave para o seu sucesso no mercado de trabalho. Prepare-se para uma jornada fascinante em que suas habilidades em grafos serão constantemente demandadas e valorizadas!

VAMOS PRATICAR

1. Grafo é uma estrutura $G = (V, E)$, em que V é um conjunto finito não nulo de vértices (ou nós), e E é um conjunto de arestas (ou arcos). Uma aresta é um par de vértices $a = \{v, w\}$, em que $v \in V$ e $w \in V$.

Dados os seguintes conjuntos $G = (V, E)$:

- a) $V = \{a\}, E = \{\emptyset\}$
- b) $V = \{a, b\}, E = \{(a, b)\}$
- c) $V = \{a, b\}, E = \langle b, a \rangle$
- d) $V = \{a, b, c\}, E = \{(a, b), (b, c)\}$
- e) $V = \{a, b, c\}, E = \langle a, b \rangle, \langle c, b \rangle, \langle a, c \rangle, \langle c, a \rangle$
- f) $V = \{a, b, c, d\}, E = \langle a, b \rangle, \langle d, a \rangle$
- g) $V = \{a, b, c, d\}, E = \{(a, b), (a, c), (a, d)\}$

Desenhe a representação em grafo de cada conjunto $G = (V, E)$.

2. Grafo é uma estrutura $G = (V, E)$, em que V é um conjunto finito não nulo de vértices (ou nós), e E é um conjunto de arestas (ou arcos). Uma aresta é um par de vértices $a = \{v, w\}$, em que $v \in V$ e $w \in V$.

Dados os seguintes conjuntos $G = (V, E)$:

- a) $V = \{a\}, E = \{\emptyset\}$
- b) $V = \{a, b\}, E = \{(a, b)\}$
- c) $V = \{a, b\}, E = \langle b, a \rangle$
- d) $V = \{a, b, c\}, E = \{(a, b), (b, c)\}$
- e) $V = \{a, b, c\}, E = \langle a, b \rangle, \langle c, b \rangle, \langle a, c \rangle, \langle c, a \rangle$
- f) $V = \{a, b, c, d\}, E = \langle a, b \rangle, \langle d, a \rangle$
- g) $V = \{a, b, c, d\}, E = \{(a, b), (a, c), (a, d)\}$

Classifique cada um dos conjuntos $G = (V, E)$ como sendo orientado, não orientado, conexo e desconexo.

3. Os grafos encontram aplicação em diversas áreas, desde algoritmos de busca até redes sociais e otimização de rotas. Seu poder de modelagem e resolução de problemas complexos continua a desempenhar um papel crucial no avanço da ciência da computação e das tecnologias relacionadas.

VAMOS PRATICAR

Qual é a definição correta de um grafo?

- a) Um grafo é uma estrutura de dados linear usada para armazenar elementos de maneira organizada.
 - b) Um grafo é um tipo de árvore binária onde todos os nós têm no máximo dois filhos.
 - c) Um grafo é um conjunto de vértices conectados por arestas, representando relações entre os vértices.
 - d) Um grafo é uma sequência ordenada de elementos com uma relação de ordem total entre eles.
 - e) Um grafo é uma estrutura de dados hierárquica usada para armazenar dados em uma organização em forma de árvore.
4. A teoria dos grafos estuda as relações entre objetos, representados por nós (vértices) e as conexões entre eles, chamadas arestas. Um dos conceitos centrais nessa área é o de “caminho”. Um caminho em um grafo é uma sequência de vértices, onde cada vértice é conectado ao próximo por uma aresta. A análise de caminhos em grafos tem implicações significativas em diversas aplicações, desde redes de transporte até redes sociais e algoritmos de busca na web.

Considere um grafo não direcionado que representa uma rede de amizades em uma rede social, onde os vértices são pessoas e as arestas indicam amizades mútuas. Se houver um caminho entre dois vértices quaisquer neste grafo, o que isso significa?

- a) Existe uma relação de inimizade entre essas duas pessoas.
- b) As duas pessoas têm um amigo em comum.
- c) Um vértice é o chefe do outro vértice na hierarquia.
- d) Essas duas pessoas não podem se comunicar na rede social.
- e) Uma pessoa está bloqueada pela outra na rede social.

VAMOS PRATICAR

5. Na teoria dos grafos, a distinção entre grafos conexos e desconexos é fundamental para compreender as propriedades e a estrutura das relações entre vértices. A distinção entre esses dois tipos de grafos tem implicações significativas em várias aplicações. Grafos conexos são, frequentemente, usados para representar redes de comunicação, redes de transporte e redes sociais, onde a conectividade entre entidades é essencial para a funcionalidade do sistema. Por outro lado, grafos desconexos podem ser usados para modelar sistemas independentes, como sistemas de informação não relacionados ou grupos distintos de elementos.

Com base nas informações apresentadas, avalie as asserções a seguir e a relação proposta entre elas:

- I - Um grafo desconexo é aquele em que não existe nenhum caminho entre pelo menos dois de seus vértices.

PORQUE

- II - Um grafo conexo é sempre um grafo com apenas um componente.

A respeito dessas asserções, assinale a opção correta:

- a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
- b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
- c) A asserção I é uma proposição verdadeira e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa e a II é uma proposição verdadeira.
- e) As asserções I e II são proposições falsas.

REFERÊNCIAS

OLIVEIRA, P. M. de; PEREIRA, R. de L. **Estrutura de Dados I**. Maringá: Unicesumar, 2019. 152 p.

TANENBAUM, A. S. **Redes de computadores**. Tradução de Vandenberg D. de Souza. Rio de Janeiro: Elsevier, 2003.

GABARITO

1.

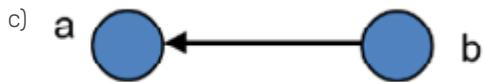
a)



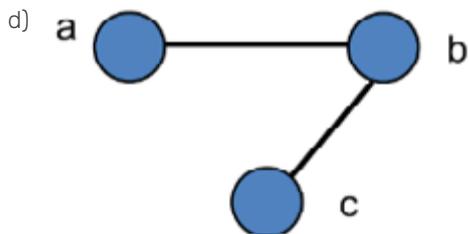
b)



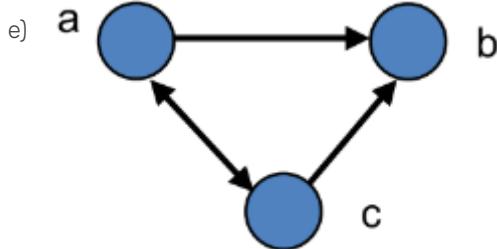
c)



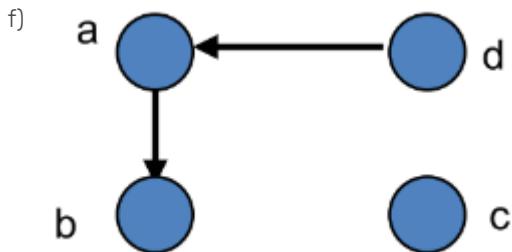
d)



e)

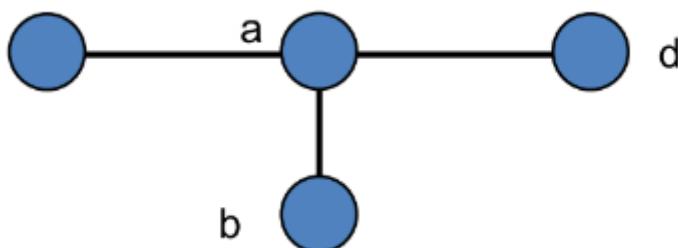


f)



GABARITO

g)



2. a) $V=\{a\}$, $E=\{\emptyset\}$

Conexo, não orientado. Também conhecido como grafo trivial por conter apenas um vértice e nenhuma aresta.

- b) $V=\{a,b\}$, $E=\{(a,b)\}$

Conexo, não orientado.

- c) $V=\{a,b\}$, $E=\{<b,a>\}$

Conexo, orientado.

- d) $V=\{a,b,c\}$, $E=\{(a,b), (b,c)\}$

Conexo, não orientado.

- e) $V=\{a,b,c\}$, $E=\{<a,b>, <c,b>, <a,c>, <c,a>\}$

Conexo, orientado.

- f) $V=\{a,b,c,d\}$, $E=\{<a,b>, <d,a>\}$

Desconexo, orientado.

- g) $V=\{a,b,c,d\}$, $E=\{(a,b), (a,c), (a,d)\}$

Conexo, não orientado.

3. **Opção C.** A teoria dos grafos é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. Para isso, são empregadas estruturas chamadas de grafos. O grafo é formado por dois conjuntos: um de vértices que não pode ser nulo e um de arestas. As arestas são formadas sempre por dois vértices que existem no conjunto e não há nenhuma obrigatoriedade da existência de arestas no grafo.

GABARITO

4. **Opção B.** Nesse contexto, o grafo representa uma rede social em que os vértices são pessoas e as arestas indicam amizades mútuas. Quando há um caminho entre dois vértices nesse grafo, isso implica que é possível percorrer uma sequência de conexões de amizades entre essas duas pessoas. Em outras palavras, existe uma rota de amizades que liga essas pessoas. Se houver um caminho entre dois vértices quaisquer (duas pessoas quaisquer) no grafo, isso significa que elas têm pelo menos um amigo em comum, já que um caminho é uma sequência de vértices conectados por arestas.

5. **Opção C.** A asserção I é verdadeira. Um grafo desconexo é aquele em que existem componentes isolados, ou seja, grupos de vértices que não têm caminhos que os conectem a outros grupos de vértices. Portanto, não existe nenhum caminho entre pelo menos dois de seus vértices. A asserção II é falsa. Um grafo conexo é aquele em que há um caminho entre qualquer par de vértices. No entanto, um grafo conexo pode ter mais de um componente (subgrafos conectados separadamente). Portanto, a segunda asserção não é verdadeira, uma vez que um grafo conexo não é necessariamente um grafo com apenas um componente.



BUSCA EM GRAFOS

MINHAS METAS

- Conhecer o algoritmo de busca em profundidade.
- Conhecer o algoritmo de busca em largura.
- Conhecer o algoritmo de Dijkstra.
- Conhecer a aplicação dos algoritmos de busca em grafos.
- Aprender a implementar os algoritmos de busca em grafos em linguagem C.

INICIE SUA JORNADA

Estudante, a busca em grafos é um conceito fundamental no campo da Ciência da Computação e da Teoria dos Grafos. Ela envolve a exploração sistemática de estruturas de grafos para encontrar informações, identificar padrões ou resolver problemas específicos.

A busca em grafos desempenha um papel crucial em diversas aplicações do mundo real, desde encontrar o caminho mais curto em um sistema de transporte até a navegação de robôs em ambientes desconhecidos.

Veremos, neste tema de aprendizagem, os dois principais algoritmos de busca em grafos que são a **base para técnicas mais complexas e eficientes**. Veremos, também, o funcionamento do algoritmo de Dijkstra, que é uma das melhores soluções de busca de caminhos de menor custo em grafos.

Lembre-se de que, se um problema pode ser modelado em um grafo, existem diversos algoritmos prontos que podem ser utilizados ou adaptados para solucioná-lo!

PLAY NO CONHECIMENTO

Vamos desbravar os caminhos por meio da busca em grafos no podcast *Desbravando Caminhos: Busca em Grafos e suas Aplicações Práticas no Dia a Dia*. Nele, vamos descobrir o que é a busca em grafos, como ela funciona e como é aplicada em situações do mundo real para resolver problemas complexos. Dá o play! **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

DESENVOLVA SEU POTENCIAL

BUSCA EM GRAFOS

Um **grafo** é uma estrutura formada por, pelo menos, um ou mais vértices (nós) e por um conjunto de arestas (arcos), que, por sua vez, pode ser vazio. Cada aresta liga dois nós do grafo. Nos algoritmos de busca que veremos, o grafo precisa ser conexo, ou seja, a partir de um nó qualquer é possível navegar por suas arestas visitando todos os demais vértices.

Muitos problemas podem ser descritos por meio de grafos, nos quais a solução para o problema requer que realizemos uma busca pelo grafo. As buscas, em geral, partem de um nó inicial em direção a um nó alvo, fazendo com que tenhamos que percorrer toda uma sequência ordenada de nós e arestas. Além disso, o próprio caminho, em si, pode ser objeto da busca, isto é, às vezes, a solução reside no caminho percorrido, e não em um nó alvo específico.

Neste tema, serão abordados, em detalhes, os métodos de busca em largura e busca em profundidade. Para isso, nos **códigos-fonte** de exemplo, levaremos em conta alguns preceitos:

INCLUSÃO

Das bibliotecas *stdlib.h*, *stdio.h* e *stdbool.h*, para auxiliar o desenvolvimento.

A DECLARAÇÃO

a declaração da constante *MAXV* para indicar o tamanho do nosso grafo (número máximo de vértices).

A DEFINIÇÃO

Do registro *str_no*, que representa um vértice.

A DECLARAÇÃO

Do vetor de structs *grafo [MAXV]*, que é o grafo em si.

Assim, antes de partirmos para os códigos-fonte dos programas, é preciso considerar tais pré-requisitos, os quais podem ser visualizados no código-fonte parcial que se segue.

```
//Bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

//Número máximo de vértices
#define MAXV 8

//Estrutura de um nó
typedef struct str_no {
    int id;
    struct str_no *proximo;
} str_no;

//Grafo
struct str_no grafo[MAXV];
```

Quadro 1 - Programa 5.1 - Inclusões de biblioteca e declarações globais para implementação das buscas em profundidade e largura / Fonte: Oliveira, Pereira (2019, p. 116).

BUSCA EM PROFUNDIDADE

O primeiro método de busca que veremos é a **busca em profundidade**. Essa técnica faz com que todo um segmento do grafo seja visitado até o final, antes que uma nova porção seja investigada. O programa, a seguir, mostra um algoritmo, em linguagem C, capaz de executar a técnica de busca em profundidade.

```
1 void buscaEmProfundidade(struct str_no g[], int inicio,
int alvo) {
2     int pilha[MAXV]; //pilha
3     bool visitado[MAXV]; //nós visitados
4     int indice = 0; //índice do topo da pilha
5     bool achou = false; //flag de controle (não visitados)
6     int corrente = inicio;
7     struct str_no *ptr;
8     int i;
9     printf("===== Busca em Profundidade =====\n");
10    //Marcando os nós como 'não visitados'.
```

```
11     for(i=0; i < MAXV; i++) {
12         visitado[i] = false;
13     }
14     while(true) {
15         //Nó corrente não visitado? Marque como visitado.
16         //Empilhe o nó corrente.
17         if(!visitado[corrente]){
18             printf("VISITANDO: %d. \n", corrente);
19             if(corrente == alvo)
20             {
21                 printf("Alvo encontrado!\n\n\n");
22                 return;
23             }
24             visitado[corrente] = true;
25             pilha[indice] = corrente;
26             indice++;
27         }
28         //Buscando por nós adjacentes, não visitados.
29         achou = false;
30         for(ptr = g[corrente].proximo; ptr != NULL;
31             ptr = ptr->proximo) {
32             if(!visitado[ptr->id]){
33                 achou = true;
34                 break;
35             }
36             if(achou) {
37                 //Atualizando o nó corrente.
38                 corrente = ptr->id;
39             }
40         else{
41             //Não há vértices adjacentes não visitados.
42             //Tentando desempilhar o vértice do topo.
43             indice--;
44             if(indice===-1){
```

```

45         //Não há mais vértices não visitados.
46         printf("Encerrando a busca. \n");
47         break;
48     }
49     corrente = pilha[indice-1];
50 }
51 }
52 return;
53 }
```

Quadro 2 - Função busca em profundidade

Fonte: Oliveira, Pereira (2019, p. 118-119).

A partir de um primeiro nó, o algoritmo coloca todos os vértices adjacentes em uma pilha e marca o nó atual como visitado. Em seguida, o programa pega o nó do topo, desempilhando-o, e repete o processo. A busca segue até que o alvo seja encontrado ou que a pilha esteja vazia.

A função busca em profundidade recebe três parâmetros. O primeiro, *g*, é a lista de adjacências que representa o grafo. O segundo parâmetro é o início, nó inicial de onde a busca partirá em direção ao terceiro parâmetro que, por sua vez, representa o *alvo* da busca. Além disso, o código-fonte faz referência à constante *MAXV*, declarada no primeiro programa, que indica o número máximo de vértices que nosso grafo pode representar.

Inicialmente, durante a execução do programa, temos a pilha vazia e, então, empilhamos o vértice inicial de partida da pesquisa. Temos uma variável *índice*, que é usada para controlar o fluxo na pilha. O laço de repetição da linha 14 se repete enquanto houver itens empilhados ou até que a busca seja concluída com sucesso.

Verificamos se o nó corrente é o alvo e, se for, imprimimos uma mensagem de sucesso na tela, encerrando a função busca em profundidade com o *return* da linha 22. Caso contrário, o algoritmo visita os nós adjacentes ao nó atual e coloca-os na pilha. Acontece, aí, o processo de desempilhamento, no qual o último nó adjacente visitado será o ponto de partida para a próxima rodada da pesquisa. Isso faz com que a busca percorra um caminho no grafo até encontrar um nó que não tenha mais vértices adjacentes.

Nesse momento, o algoritmo inicia uma nova busca a partir do último nó empilhado. Observe o comportamento na Figura 1:

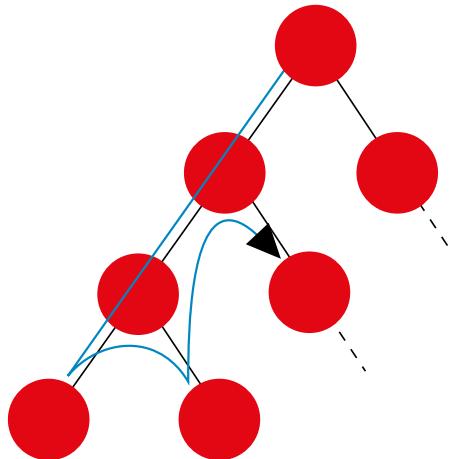


Figura 1 - Exemplo de busca em profundidade

Fonte: Oliveira, Pereira (2019, p. 119).

Descrição da Imagem: a figura tem sete bolas vermelhas com linhas ligando-as e quatro níveis. A primeira está posicionada sozinha no primeiro nível, com duas abaixo dela, uma à esquerda e outra à direita, ligadas à primeira. Outras duas estão abaixo da segunda no terceiro nível e conectadas a ela repetindo a mesma formação no quarto nível. Uma linha azul começa na primeira bola e passa por cima das mais à esquerda até o último nível depois retorna passando pelas da direita parando e apontando para a bola da direita do terceiro nível. Ficando apenas a bola da direita do segundo nível sem passar pela linha azul. Fim da descrição.

BUSCA EM LARGURA

A **busca em largura** se assemelha à busca em profundidade, estudada recentemente. A principal diferença é que os nós visitados são enfileirados, em vez de empilhados. Isso garante, primeiramente, que sejam percorridos todos os nós adjacentes ao nó atual para, só então, visitar os nós mais distantes, repetindo o processo.

A pesquisa foi iniciada no nó 1, usando o algoritmo de busca em largura. Ele irá visitar primeiro o nó 2, depois o nó 3, o nó 4, e assim por diante. Observe a Figura 2:

Os nós visitados são enfileirados, em vez de empilhados

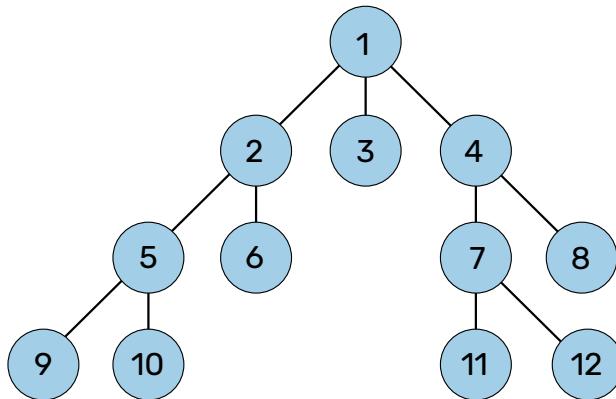


Figura 2 - Ordem que os nós são pesquisados na busca em largura
Fonte: Oliveira, Pereira (2019, p. 120).

Descrição da Imagem: a figura tem quatro níveis, um abaixo do outro, com círculos com números representando os nós. No primeiro nível está o nó um. No segundo nível está no nó dois, três e quatro, todos ligados ao nó um. No terceiro nível temos os nós cinco e seis ligado ao nó dois e os nós sete e oito ligados ao nó quatro. No nível quatro, temos os nós nove e dez ligados ao nó cinco e os nós onze e doze ligados ao nó sete. Fim da descrição.

A **busca em largura**, quando aplicada ao grafo da Figura 2, resultaria na seguinte ordem de visitação: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 e 12. Por outro lado, se fôssemos aplicar a busca em profundidade no mesmo grafo, começando pelo nó 1, a sequência seria: 1, 2, 5, 9, 10, 6, 3, 4, 7, 11, 12 e 8. **Um resultado bem diferente, não é mesmo?**

O Quadro 3, a seguir, traz a função busca em profundidade, implementada em linguagem C. Essa função recebe os mesmos parâmetros da função busca em profundidade.

Seu funcionamento é praticamente o mesmo que o da função busca em profundidade, com a diferença de que empregamos uma lista para guiar a ordem de visitação dos nós, em vez de uma pilha.

```

void buscaEmLargura(struct str_no g[], int inicio,
int alvo){
    int fila[MAXV]; //fila
    bool visitado[MAXV]; //nós visitados
    int indice = 0; //controle da fila
    bool achou = false; //flag (não visitados)
    int corrente = inicio;
    struct str_no *ptr;
    int i;
    printf("===== Busca em Largura ===== \n");

    //Marcando os nós como 'não visitados'.
    for(i=0; i < MAXV; i++) {
        visitado[i] = false;
    }

    //Partindo do primeiro vértice.
    printf("VISITANDO: %d. \n", corrente);
    visitado[corrente] = true;
    fila[indice] = corrente;
    indice++;
    while(true){
        //Visitar os nós adjacentes ao vértice
        corrente
        for(ptr = g[corrente].proximo; ptr != NULL; ptr = ptr->proximo){
            //Caso corrente ainda não tenha
            sido visitado:
            corrente = ptr->id;
            if(!visitado[corrente]){
                //Enfileira e marca como visi-
                tado.
                printf("VISITANDO: %d. \n",
                corrente);
                if(corrente == alvo)

                {

                    printf("Alvo encontra-
                    do!\n\n\n");
                    return;
                }
            }
            visitado[corrente] = true;
        }
    }
}

```

```
        fila[indice] = corrente;
        indice++;

    }

}

//Caso a fila não esteja vazia:
if(indice!=0)
{
    //Atualizando vértice corrente.
    corrente = fila[0];
    //Desenfileirando o primeiro vértice.
    for(i=1;i<indice+1;i++) {
        fila[i-1]=fila[i];
    }
    indice--;
}
else
{
    //Não há mais vértices para visitar.
    printf("Encerrando busca.\n");

    break;
}
return;
}
```

Quadro 3 - Programa 5.3 - Função busca em largura
Fonte: Oliveira, Pereira (2019, p. 122).



PENSANDO JUNTOS

Dois algoritmos praticamente idênticos com apenas um pequeno detalhe. Um usa uma estrutura em pilha para guardar os nós visitados e o outro, a estrutura de fila. Esse pequeno detalhe faz com que a busca seja totalmente diferente.

ALGORITMO DE DIJKSTRA

Em 1956, o cientista da computação holandês Edsger Dijkstra concebeu um algoritmo que, em sua homenagem, passou a ser chamado de **algoritmo de Dijkstra**. Sua publicação ocorreu em 1958 e tem como objetivo **solucionar o problema do caminho mais curto entre dois vértices em grafos conexos com arestas de pesos não negativos**.

O algoritmo de Dijkstra assemelha-se ao de busca em largura que acabamos de estudar, mas é considerado um **algoritmo guloso**, ou seja, **toma a decisão que parece ótima no momento**. A estratégia gulosa é muito interessante ao se tratar de problemas complexos ou para análises em grandes quantidades de dados.

Imagine um GPS que precisa buscar o caminho do ponto onde você está e um determinado endereço. Internamente, o mapa das ruas é armazenado em forma de grafos e, para achar o caminho entre dois pontos, basta realizar uma busca no grafo.

Todavia, como praticamente todas as ruas de uma região são interligadas, a quantidade de caminhos possíveis será muito grande, por isso ignorar caminhos de custo inicial elevado ajuda a diminuir de forma significativa a região de busca da melhor solução.

O algoritmo de Dijkstra leva em consideração uma matriz de custos. Cada entrada na matriz tem armazenado o custo (peso) da aresta entre dois vértices. Durante a visita aos vértices adjacentes, o programa inclui na fila apenas os vértices de menor custo.

O Quadro 4, a seguir traz a implementação completa do algoritmo de Dijkstra. Ele tem um menu de opções, permite a criação de um grafo e o peso de suas arestas. Ele calcula os caminhos mais curtos entre quaisquer dois pontos no grafo.



```
//Bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Variáveis
int destino, origem, vertices = 0;
int custo, *custos = NULL;

//Prototipação
void dijkstra(int vertices, int origem, int destino, int
*custos);
void menu_mostrar(void);
void grafo_procurar(void);
void grafo_criar(void);

//Função principal
int main(int argc, char **argv) {
    int opt = -1;
    //Laço principal do menu
    do {
        //Desenha o menu na tela
        menu_mostrar();
        scanf("%d", &opt);
        switch (opt) {
            case 1:
                //cria um novo grafo
                grafo_criar();
                break;
            case 2:
                //procura os caminhos
                if (vertices > 0) {
                    grafo_procurar();
                }
                break;
        }
    } while (opt != 0);
    printf("\nAlgoritmo de Dijkstra finaliza-
do...\n\n");
    system("pause");
    return 0;
}

//Implementação do algoritmo de Dijkstra
void dijkstra(int vertices, int origem, int destino, int
```

```
*custos)
{
    int i, v, cont = 0;
    int *ant, *tmp;
    int *z; /* vertices para os quais se conhece o
caminho minimo */
    double min;
    double dist[vertices]; /* vetor com os custos
dos caminhos */
    /* aloca as linhas da matriz */
    ant = (int*) calloc (vertices, sizeof(int *));
    if (ant == NULL) {
        system("color fc");
        printf ("** Erro: Memoria Insuficiente **");
        exit(-1);
    }
    tmp = (int*) calloc (vertices, sizeof(int *));
    if (tmp == NULL) {
        system("color fc");
        printf ("** Erro: Memoria Insuficiente **");
        exit(-1);
    }
    z = (int *) calloc (vertices, sizeof(int *));
    if (z == NULL) {
        system("color fc");
        printf ("** Erro: Memoria Insuficiente **");
        exit(-1);
    }
    for (i = 0; i < vertices; i++) {
        if (custos[(origem - 1) * vertices + i] != - 1) {
            ant[i] = origem - 1;
            dist[i] = custos[(origem-1)*vertices+i];
        }
        else {
            ant[i]= -1;
            dist[i] = HUGE_VAL;
        }
        z[i]=0;
    }
    z[origem-1] = 1;
    dist[origem-1] = 0;
    /* Laco principal */
    do {
        /* Encontrando o vertice que deve entrar em z */
        min = HUGE_VAL;
```

```
for (i=0;i<vertices;i++) {
    if (!z[i]) {
        if (dist[i]>=0 && dist[i]<min) {
            min=dist[i];v=i;
        }
    }
}
/* Calculando as distancias dos novos vizinhos
de z */
if (min != HUGE_VAL && v != destino - 1) {
    z[v] = 1;
    for (i = 0; i < vertices; i++) {
        if (!z[i]) {
            if (custos[v*vertices+i] != -1 && dist[v]
+ custos[v*vertices+i] < dist[i]) {
                dist[i] = dist[v] + custos[v*vertices+i];
                ant[i] =v;
            }
        }
    }
}
} while (v != destino - 1 && min != HUGE_VAL);
/* Mostra o Resultado da busca */
printf("\tDe %d para %d: \t", origem, destino);
if (min == HUGE_VAL) {
    printf("Nao Existe\n");
    printf("\tCusto: \t- \n");
}
else {
    i = destino;
    i = ant[i-1];
    while (i != -1) {
        tmp[cont] = i+1;
        cont++;
        i = ant[i];
    }
    for (i = cont; i > 0 ; i--) {
        printf("%d -> ", tmp[i-1]);
    }
    printf("%d", destino);
    printf("\n\tCusto: %d\n", (int) dist[destino-1]);
}
}
```

```

void grafo_criar(void) {
    do {
        printf("\nInforme o numero de vertices (no
minimo 3 ): ");
        scanf("%d", &vertices);
    } while (vertices < 3 );
    if (!custos) {
        free(custos);
    }
    custos = (int *) malloc(sizeof(int)*verti-
ces*vertices);
    //Se o compilador falhou em alocar espaço na
memória
    if (custos == NULL) {
        system("color fc");
        printf ("** Erro: Memoria Insuficiente
**");
        exit(-1);
    }
    //Preenchendo a matriz com -1
    for (int i = 0; i <= vertices * vertices; i++) {
        custos[i] = -1;
    }
    do {
        system("cls");
        printf("Entre com as Arestas:\n");
        do {
            printf("Origem (entre 1 e %d ou '0'
para sair): ", vertices);
            scanf("%d", &origem);
        } while (origem < 0 || origem > vertices);
        if (origem) {
            do {
                printf("Destino (entre 1 e %d,
menos %d): ", vertices,
origem);
                scanf("%d", &destino);
            } while (destino < 1 || destino >
vertices || destino == origem);
            do {
                printf("Custo (positivo) do
vertice %d para o vertice %d:
",
origem, destino);
            }
        }
    }
}

```

```

        scanf("%d", &custo);
    } while (custo < 0);
    custos[(origem-1) * vertices + destino - 1] = custo;
}
} while (origem);
}

//Busca os menores caminhos entre os vértices
void grafo_procurar(void){
    int i, j;
    system("cls");
    system("color 03");
    printf("Lista dos Menores Caminhos no Grafo
Dado: \n");
    for (i = 1; i <= vertices; i++) {
        for (j = 1; j <= vertices; j++) {
            dijkstra(vertices, i,j, custos);
        }
        printf("\n");
    }
    system("pause");
    system("color 07");
}
//Desenha o menu na tela
void menu_mostrar(void){
    system("cls");
    printf("Implementacao do Algoritmo de Dijask-
tra\n");
    printf("Opcoes:\n");
    printf("\t 1 - Adicionar um Grafo\n");
    printf("\t 2 - Procura Os Menores Caminhos no
Grafo\n");
    printf("\t 0 - Sair do programa\n");
    printf("? ");
}
}

```

Quadro 4 - Programa 5.4 - Algoritmo de Dijkstra em C

Fonte: Oliveira, Pereira (2019, p. 123-128).

**EU INDICO**

Este vídeo oferece uma visão envolvente da vida e do trabalho de Edsger Dijkstra, um dos pioneiros na área da Ciência da Computação. Dijkstra é conhecido por suas contribuições significativas para a teoria dos grafos, algoritmos de busca e programação estruturada, que moldaram os fundamentos da computação moderna. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

**EU INDICO**

Eu separei para você a parte principal do algoritmo de Dijkstra! A variável `min` irá guardar o menor valor encontrado. Ela é inicializada com um valor muito grande. A constante `HUGE_VAL` está presente na biblioteca `math.h` e foi criada para essa finalidade.

Em seguida, existe um laço de repetição em que é verificado se a distância entre o vértice atual e o vértice adjacente é menor do que o contido na variável `min`. Se for, o valor da variável é atualizado e o processo se repete até que o vértice não tenha mais nós adjacentes.

```
min = HUGE_VAL;
for (i=0;i<vertices;i++) {
    if (!z[i]){
        if (dist[i]>=0 && dist[i]<min) {
            min=dist[i];v=i;
        }
    }
}
```

Quadro 5 - Programa 5.5 - O coração de Dijkstra
Fonte: Oliveira, Pereira (2019, p. 128).

Na prática, como se aplica? Digamos que você trabalhe para uma empresa aérea. Cada aeroporto é um nó e cada voo entre dois aeroportos é uma aresta. Você pode possuir uma tabela na qual conste o preço da passagem entre dois trechos; ou ainda existir uma segunda tabela que tenha a duração dos voos entre os aeroportos das cidades. No balcão de venda, o algoritmo de Dijkstra pode procurar a melhor rota entre a origem e o destino levando em consideração o valor da passagem ou o tempo de duração do voo.

Imagine a aplicação dessa técnica no ramo da logística, no roteamento de pacotes numa rede comercial ou na escolha do fornecedor com melhor relação custo por tempo de entrega.

O algoritmo de Dijkstra é muito utilizado em situações em que é preciso minimizar custos ou otimizar recursos.

Neste tema de aprendizagem, vimos os dois principais métodos de busca em grafos, a busca em profundidade e a busca em largura. Ambas as técnicas são simples e bem parecidas, alterando apenas a forma como os vetores visitados são armazenados.

Esses algoritmos são a base para praticamente todas as demais soluções para a busca de caminho em grafos conexos. O **próprio algoritmo de Dijkstra**, que tivemos a oportunidade de estudar, é um algoritmo de busca em largura modificado para uma solução específica, que é encontrar o menor caminho entre dois vértices para grafos que possuem pesos nas arestas.

Em resumo, a busca em grafos é uma ferramenta poderosa e versátil para resolver uma ampla variedade de problemas em Ciência da Computação, inteligência artificial e jogos. Ela ajuda a encontrar soluções ótimas ou aproximadas, explorar cenários possíveis e navegar por estruturas complexas de maneira eficiente. O conhecimento dessas técnicas é muito importante, pois, se um problema puder ser modelado na forma de um grafo, existem diversos algoritmos que podem ser utilizados ou adaptados para encontrar a solução.

EM FOCO

Assista agora ao conteúdo referente a sua aula. **Recursos de mídia disponíveis no conteúdo digital do ambiente virtual de aprendizagem.**

NOVOS DESAFIOS

Estudante, a aplicação de algoritmos de busca em grafos, muitas vezes, envolve a resolução de problemas complexos e a criação de soluções eficientes. O campo de algoritmos de busca em grafos está em constante evolução, com novas técnicas e tecnologias surgindo regularmente.

O trabalho na aplicação de algoritmos de busca em grafos pode ter um impacto tangível na eficiência operacional, na tomada de decisões estratégicas e na resolução de problemas do mundo real.

Você pode encontrar oportunidades em empresas de tecnologia, empresas de transporte e logística, provedores de telecomunicações, instituições acadêmicas, institutos de pesquisa e, até mesmo, em empresas farmacêuticas e de biotecnologia. O setor público, incluindo agências governamentais de transporte e saúde, também oferece oportunidades.

Em resumo, o profissional de aplicação de algoritmos de busca em grafos tem um mercado de trabalho em crescimento e diversificado à sua disposição. Esteja preparado para abraçar desafios complexos e contribuir para soluções que impactam positivamente nossa sociedade cada vez mais interconectada.

VAMOS PRATICAR

1. A busca em grafos é uma estratégia fundamental utilizada para explorar e analisar as conexões entre vértices e as estruturas presentes nos grafos. Duas abordagens principais são a busca em profundidade e a busca em largura, ambas aplicadas para percorrer grafos de maneira sistemática. Esses algoritmos de busca têm amplas aplicações em diversas áreas, como redes sociais, roteamento de redes, otimização, jogos e muitas outras. Eles também servem como blocos de construção essenciais para outros algoritmos, como o algoritmo de Dijkstra para encontrar caminhos mais curtos e o algoritmo de busca em largura para árvores em estruturas hierárquicas.

Qual é a principal diferença entre a busca em largura e a busca em profundidade?

2. A busca em grafos é uma estratégia fundamental utilizada para explorar e analisar as conexões entre vértices e as estruturas presentes nos grafos. Duas abordagens principais são a busca em profundidade e a busca em largura, ambas aplicadas para percorrer grafos de maneira sistemática. Esses algoritmos de busca têm amplas aplicações em diversas áreas, como redes sociais, roteamento de redes, otimização, jogos e muitas outras. Eles também servem como blocos de construção essenciais para outros algoritmos, como o algoritmo de Dijkstra para encontrar caminhos mais curtos e o algoritmo de busca em largura para árvores em estruturas hierárquicas.

Em que caso a busca em largura é mais eficiente que a busca em profundidade?

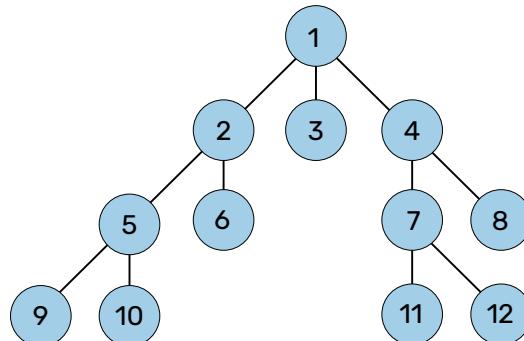
3. O algoritmo de busca em profundidade é uma técnica amplamente estudada na teoria dos grafos e na ciência da computação. Ele foi introduzido como uma estratégia de exploração de grafos por Charles Pierre Trémaux no século XIX. Desde então, o algoritmo de busca em profundidade tem sido uma ferramenta essencial em algoritmos e estruturas de dados, desempenhando um papel crucial na resolução de problemas práticos.

Qual é o objetivo principal do algoritmo de busca em profundidade em grafos?

- f) Encontrar o caminho mais curto entre dois vértices.
- g) Encontrar o vértice com o menor grau no grafo.
- h) Encontrar todos os ciclos no grafo.
- i) Encontrar o vértice de maior grau no grafo.
- j) Explorar o máximo possível em um ramo do grafo antes de retroceder

VAMOS PRATICAR

4. Observe a figura a seguir: a pesquisa foi iniciada no nó 1, usando o algoritmo de busca em largura. Ele irá visitar primeiro o nó 2, depois o nó 3, o nó 4, e assim por diante.



A busca em largura, quando aplicada ao grafo da figura, resultaria na seguinte ordem de visitação: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.

Com isso em mente, indique qual é a principal característica do algoritmo de busca em largura em grafos.

- a) Explora o máximo possível em um ramo do grafo antes de retroceder.
 - b) Seleciona aleatoriamente vértices para explorar.
 - c) Explora primeiro os vértices de maior grau.
 - d) Explora vértices vizinhos antes de explorar vértices mais distantes.
 - e) Prioriza a exploração de vértices em ciclos.
5. Em 1956, o cientista da computação holandês Edsger Dijkstra concebeu um algoritmo que, em sua homenagem, passou a ser chamado de algoritmo de Dijkstra. O algoritmo de Dijkstra leva em consideração uma matriz de custos. Cada entrada na matriz tem armazenado o custo (peso) da aresta entre dois vértices. Durante a visita aos vértices adjacentes, o algoritmo inclui na fila apenas os vértices de menor custo.

VAMOS PRATICAR

Com base nas informações apresentadas, avalie as asserções a seguir e a relação proposta entre elas:

III - O algoritmo de Dijkstra é usado para encontrar o caminho mais curto entre dois vértices em grafos conexos.

PORQUE

IV - O algoritmo de Dijkstra é eficiente em grafos que contenham arestas com pesos negativos.

A respeito dessas asserções, assinale a opção correta:

- a) As asserções I e II são verdadeiras, e a II é uma justificativa correta da I.
- b) As asserções I e II são verdadeiras, mas a II não é uma justificativa correta da I.
- c) A asserção I é uma proposição verdadeira e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa e a II é uma proposição verdadeira.
- e) As asserções I e II são proposições falsas.

REFERÊNCIAS

OLIVEIRA, P. M. de; PEREIRA, R de L. **Estrutura de Dados I**. Maringá: Unicesumar, 2019. 152 p.

GABARITO

1. A busca em largura visita, primeiramente, os vértices mais próximos antes de se aprofundar no grafo. Já a busca em profundidade segue um caminho até o final antes de optar por uma nova ramificação. Isso se dá pela forma como os vértices visitados são armazenados. A busca em profundidade armazena os dados numa pilha e a busca em largura, em uma fila.
 2. A busca em largura é mais eficiente quando sabemos que o valor procurado está nas proximidades da região de pesquisa. A busca em profundidade pode obter um melhor resultado se o resultado da pesquisa estiver mais distante.
- 3. Opção E.** O principal objetivo do algoritmo é explorar, o máximo possível em um ramo do grafo antes de retroceder e explorar outros ramos. Começando de um vértice inicial, o algoritmo percorre as arestas de forma recursiva, visitando vértices adjacentes ainda não visitados. Esse processo é realizado até que todos os vértices sejam visitados ou até que não haja mais vértices a serem explorados a partir de uma determinada ramificação.
- 4. Opção D.** Como a busca em largura explora vértices vizinhos antes de explorar vértices mais distantes, a busca quando aplicada ao grafo da figura, resultaria na seguinte ordem de visitação: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.
- 5. Opção C.** A asserção I é verdadeira. O algoritmo de Dijkstra é amplamente conhecido por encontrar o caminho mais curto entre um vértice de origem e todos os outros vértices em grafos conexos. A asserção II é falsa. O algoritmo de Dijkstra é projetado para funcionar corretamente apenas em grafos com pesos de arestas não negativos. A presença de pesos negativos pode levar a resultados incorretos ou imprecisos ao utilizar o algoritmo de Dijkstra.

MINHAS ANOTAÇÕES