



ESTRUTURA DE DADOS I

Professor Me. Pietro Martins de Oliveira
Professor Me. Rogério de Leon Pereira



Acesse o seu livro também disponível na versão digital.

Quando identificar o ícone QR-CODE, utilize o aplicativo **Unicesumar Experience** para ter acesso aos conteúdos online. O download do aplicativo está disponível nas plataformas:



Google Play



App Store



Professor
Wilson de Matos Silva
Reitor

Em um mundo global e dinâmico, nós trabalhamos com princípios éticos e profissionalismo, não só para oferecer uma educação de qualidade, mas, acima de tudo, para gerar uma conversão integral das pessoas ao conhecimento. Baseamo-nos em 4 pilares: intelectual, profissional, emocional e espiritual.

Iniciamos a Unicesumar em 1990, com dois cursos de graduação e 180 alunos. Hoje, temos mais de 100 mil estudantes espalhados em todo o Brasil: nos quatro campi presenciais (Maringá, Curitiba, Ponta Grossa e Londrina) e em mais de 300 polos EAD no país, com dezenas de cursos de graduação e pós-graduação. Produzimos e revisamos 500 livros e distribuímos mais de 500 mil exemplares por ano. Somos reconhecidos pelo MEC como uma instituição de excelência, com IGC 4 em 7 anos consecutivos. Estamos entre os 10 maiores grupos educacionais do Brasil.

A rapidez do mundo moderno exige dos educadores soluções inteligentes para as necessidades de todos. Para continuar relevante, a instituição de educação precisa ter pelo menos três virtudes: inovação, coragem e compromisso com a qualidade. Por isso, desenvolvemos, para os cursos de Engenharia, metodologias ativas, as quais visam reunir o melhor do ensino presencial e a distância.

Tudo isso para honrarmos a nossa missão que é promover a educação de qualidade nas diferentes áreas do conhecimento, formando profissionais cidadãos que contribuam para o desenvolvimento de uma sociedade justa e solidária.

Vamos juntos!



Janes Fidélis Tomelin

Pró-Reitor de Ensino de EaD

Kátia Solange Coelho

Diretoria de Graduação e Pós

Débora do Nascimento Leite

Diretoria de Design Educacional

Leonardo Spaine

Diretoria de Permanência

Seja bem-vindo(a), caro(a) acadêmico(a)! Você está iniciando um processo de transformação, pois quando investimos em nossa formação, seja ela pessoal ou profissional, nos transformamos e, consequentemente, transformamos também a sociedade na qual estamos inseridos. De que forma o fazemos? Criando oportunidades e/ou estabelecendo mudanças capazes de alcançar um nível de desenvolvimento compatível com os desafios que surgem no mundo contemporâneo.

O Centro Universitário Cesumar mediante o Núcleo de Educação a Distância, o(a) acompanhará durante todo este processo, pois conforme Freire (1996): "Os homens se educam juntos, na transformação do mundo".

Os materiais produzidos oferecem linguagem dialógica e encontram-se integrados à proposta pedagógica, contribuindo no processo educacional, complementando sua formação profissional, desenvolvendo competências e habilidades, e aplicando conceitos teóricos em situação de realidade, de maneira a inseri-lo no mercado de trabalho. Ou seja, estes materiais têm como principal objetivo "provocar uma aproximação entre você e o conteúdo", desta forma possibilita o desenvolvimento da autonomia em busca dos conhecimentos necessários para a sua formação pessoal e profissional.

Portanto, nossa distância nesse processo de crescimento e construção do conhecimento deve ser apenas geográfica. Utilize os diversos recursos pedagógicos que o Centro Universitário Cesumar lhe possibilita. Ou seja, acesse regularmente o Studeo, que é o seu Ambiente Virtual de Aprendizagem, interaja nos fóruns e enquetes, assista às aulas ao vivo e participe das discussões. Além disso, lembre-se que existe uma equipe de professores e tutores que se encontra disponível para sanar suas dúvidas e auxiliá-lo(a) em seu processo de aprendizagem, possibilitando-lhe trilhar com tranquilidade e segurança sua trajetória acadêmica.

Professor Me. Pietro Martins de Oliveira

Mestre em Ciência da Computação na área de Visão Computacional pela Universidade Estadual de Maringá (2015). Graduado em Engenharia de Computação pela Universidade Estadual de Ponta Grossa (2011). Atuou como analista de sistemas e programador nas empresas Siemens Enterprise Communications (Centro Internacional de Tecnologia de Software - CITS, 2011-2012), e Benner Saúde Maringá (2015). Experiência de dois anos como coordenador dos cursos de Bacharelado em Engenharia de Software, Sistemas de Informação e Engenharia de Produção. Docente no ensino superior.

Professor Me. Rogério de Leon Pereira

Mestre em Ciência da Computação pela Universidade Estadual de Maringá (2006). Graduado em Tecnologia em Processamento de Dados pelo Centro de Ensino Superior de Maringá (1999). Atualmente, é analista de informática da Universidade Estadual de Maringá. Tem experiência na área de Ciência da Computação, com ênfase em desenvolvimento de sistemas Web.

SEJA BEM-VINDO(A)!

Na Unidade 1, revisaremos os conceitos de variáveis homogêneas e heterogêneas. É importante que retomemos o entendimento sobre vetores, matrizes e registros. Fazemos isso pois esses são os elementos básicos para construir estruturas de dados mais avançadas.

Além disso, ainda na Unidade 1, teremos contato com um novo conceito, o de Ponteiros. Esse que também é um conceito básico para que seja possível compreender as estruturas de dados alocadas dinamicamente.

Passando para a Unidade 2, com os conceitos de vetores e registros em mente, veremos as duas primeiras estruturas de dados diferenciadas: a Fila e a Pilha.

Aliando o conceito de ponteiros e as estruturas de dados recém mencionadas, estudaremos sobre listas dinâmicas, na Unidade 3. Versaremos sobre listas dinâmicas genéricas, listas duplamente encadeadas, listas circulares, bem como pilhas e filas alocadas de maneira dinâmica em memória.

Nas duas últimas unidades, adentraremos na abstração matemática denominada Grafo. Na Unidade 4, introduziremos o conceito de grafos e como eles podem ser modelados em nossos programas de computador.

Na Unidade 5, tendo fixado o conceito de grafos, aprenderemos a realizar buscas nesse tipo de estrutura de dados. Em especial, falaremos sobre a Busca em Largura, a Busca em Profundidade e o Algoritmo de Dijkstra.

Lembre-se que o curso de graduação é criado seguindo um processo lógico e estruturado de formação do conhecimento. Você já aprendeu sobre o funcionamento interno de uma máquina na disciplina de Arquitetura de Computadores. Mais do que isso, já estudou sobre variáveis simples e compostas, homogêneas e heterogêneas, na matéria de Algoritmos e Lógica de Programação.

O que veremos aqui neste livro é uma sequência desse conteúdo e a sua respectiva aplicação, que servirá de base para as próximas disciplinas de cunho técnico do seu curso e para a sua formação como profissional de Tecnologia da Informação.

SUMÁRIO

UNIDADE I

PONTEIROS

15	Introdução
15	Estruturas Homogêneas e Heterogêneas
16	Vetores e Matrizes
17	Registros
18	Ponteiros
26	Propriedades de Ponteiros
28	Alocação Dinâmica na Memória
33	Criando Vetores Dinâmicos
35	Considerações Finais

UNIDADE II

PILHAS E FILAS

41	Introdução
41	Pilhas
53	Filas
62	Considerações Finais



SUMÁRIO

UNIDADE III

LISTAS DINÂMICAS

-
- 69 Introdução
 - 69 Fundamentos de Listas Dinâmicas
 - 73 Implementando uma Lista Dinâmica
 - 80 Lista Dinâmica com Forma de Pilha
 - 84 Lista Dinâmica com Forma de Fila
 - 88 Considerações Finais

UNIDADE IV

GRAFOS

-
- 93 Introdução
 - 93 Sete Pontes de Königsberg
 - 96 Teoria dos Grafos
 - 98 Grafos Como Representação de Problemas
 - 99 Representação Computacional de Grafos
 - 102 Implementando Grafos em C
 - 109 Considerações Finais



SUMÁRIO

 UNIDADE V

BUSCA EM GRAFOS

115 Introdução

115 Busca em Grafos

117 Busca em Profundidade

120 Busca em Largura

122 Algoritmo de Dijkstra

129 Considerações Finais

134 REFERÊNCIAS

135 GABARITO

146 CONCLUSÃO



PONTEIROS

UNIDADE

I

Objetivos de Aprendizagem

- 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 num 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.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Estruturas homogêneas e heterogêneas
- Vetores e matrizes
- Registros
- Ponteiros
- Propriedades de ponteiros
- Alocação dinâmica na memória
- Criando vetores dinâmicos

INTRODUÇÃO

Existe uma lenda que atormenta o sono dos programadores, e o nome dela é Ponteiro. Nesta unidade, você verá de forma simples e com muitos exemplos práticos que não há necessidade de perder o seu sono por causa dos mal falados ponteiros.

Essa estrutura é uma das mais importantes, porque graças a ela é possível fazer alocação dinâmica na memória, navegar nela para 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 serão usadas ao longo de toda a disciplina.

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

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 5 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.

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 tipos diferentes.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

VETORES E MATRIZES

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.

```
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, o exemplo da declaração de uma matriz de números reais, com duas linhas e três colunas.

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

Lembre-se que são necessários dois índices para acessar os dados em uma matriz bidimensional.

```
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 tipos diferentes. 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*.

```
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 livro.

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.



Ao invés 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;

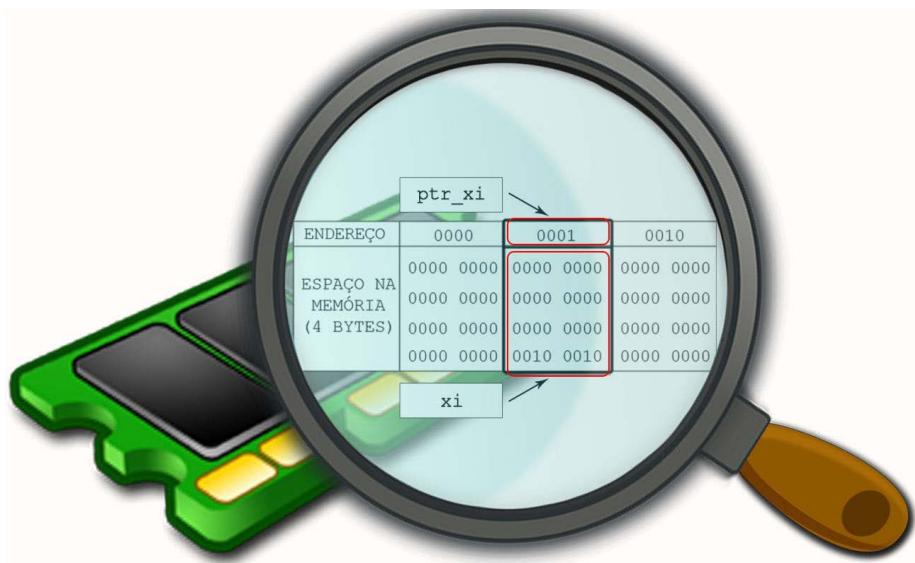
char xc;
char *ptr_xc;

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”.

A imagem a seguir simula um espaço na memória. Na parte de cima estão os endereços e, na de baixo, os valores contidos naquelas posições. Essa ilustração ajuda a entender melhor o conceito de ponteiro e a sua relação com uma variável.



Fonte: ilustração de Clarissa Brasil

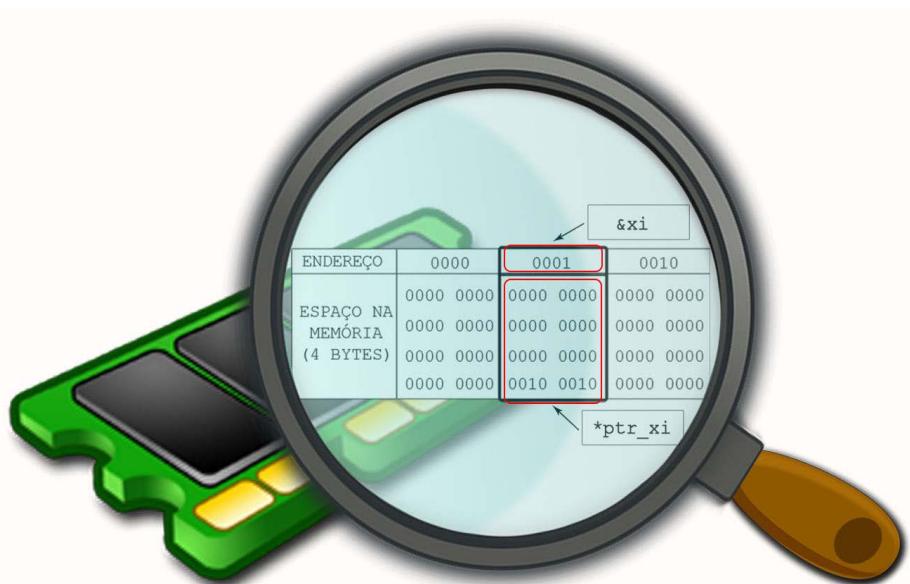
Como *ptr_xi* é um ponteiro, não posso simplesmente atribuir a ele o valor de *xi*, preciso, sim, atribuir o endereço que *xi* ocupa na memória. Para isso, usamos a anotação *&xi*, que significa “o ponteiro que aponta para o endereço na memória da variável *xi*”.

```
ptr_xi = &xi;
```

Eu sei 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.



Fonte: ilustração de Clarissa Brasil

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, posto 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 ao invés 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);  
}
```

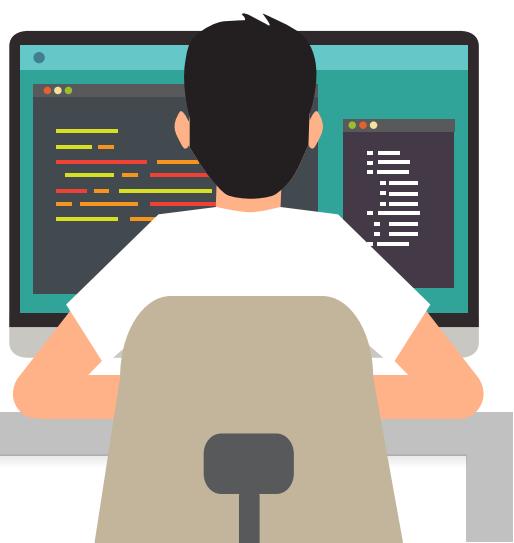
Já sabemos que o valor de xi não será 10, já que o atualizamos para 20 antes da impressão, não é? Fizemos apenas uma alteração no valor de $*ptr_xi$ e sabemos que o ponteiro ptr_xi aponta para o conteúdo de xi , não o contrário. Repare no que aconteceu agora:

```
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 alteraremos 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. E, 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 nesta parte do livro. 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);
}
```

Essa é a saída esperada 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 estamos apenas criando uma variável que aponta para um endereço, estamos criando uma variável que aponta para um endereço de um determinado tipo.

Como vimos no início do livro, 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;
```

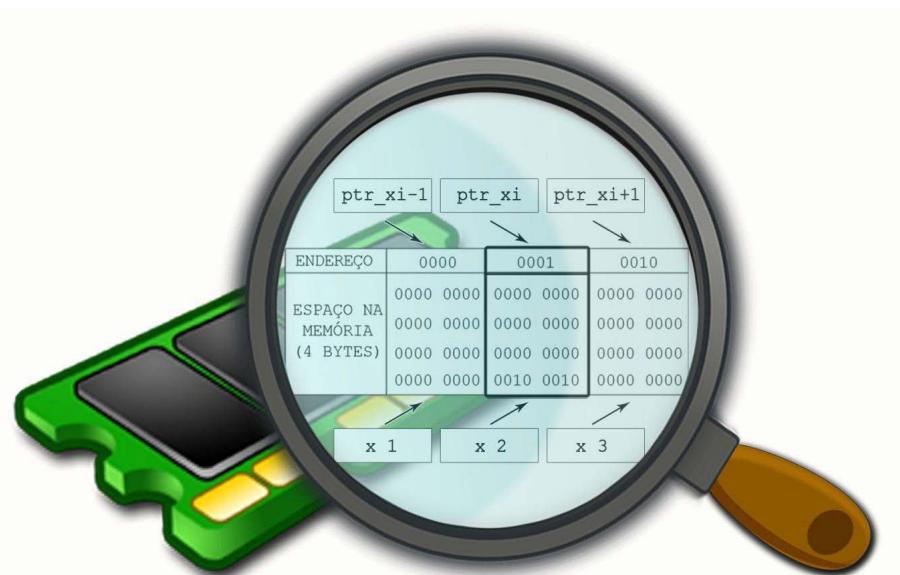
Porém é possível em C fazer a conversão de ponteiros de tipos diferentes. Por exemplo, é possível converter o valor de *ptr_xf* para o tipo **ponteiro para um inteiro** por meio do comando (*int **).

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

Pare tudo! Pare o mundo! Agora você deve estar se perguntando: “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

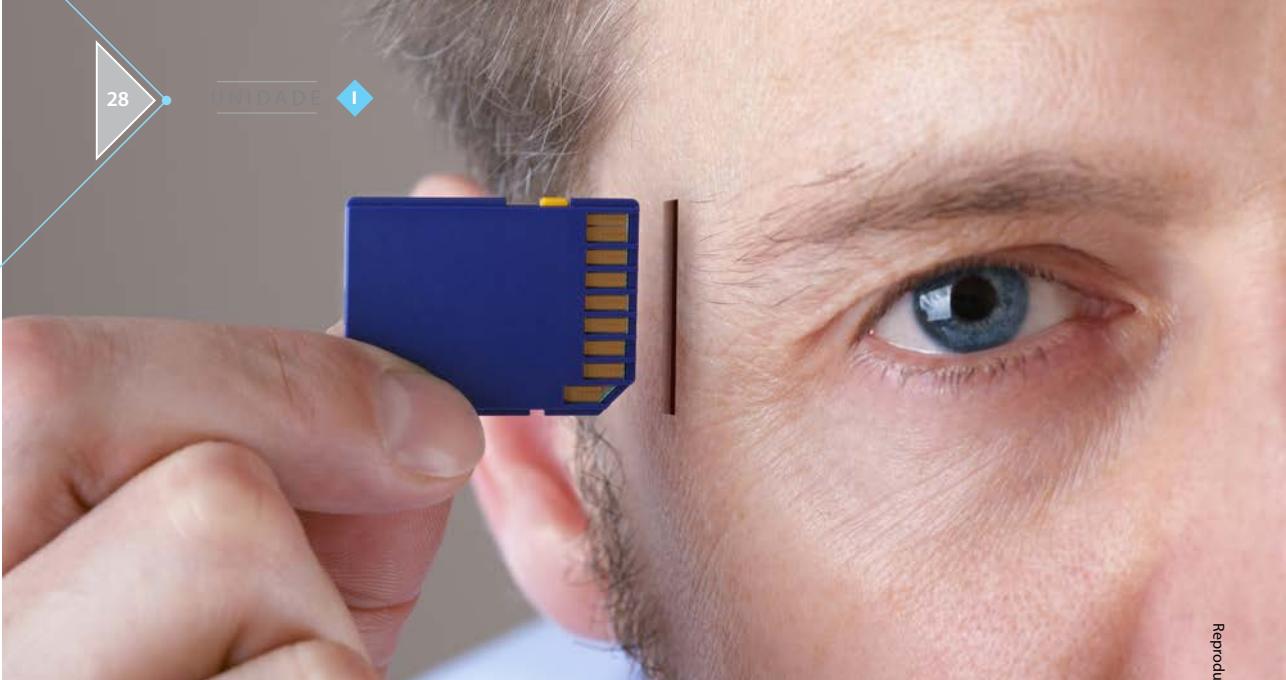
$*ptr_{xi}$, e $ptr_{xi} - 1$ irá apontar para o primeiro endereço de um número inteiro imediatamente anterior a $*ptr_{xi}$. Dessa forma, a partir de um ponteiro, é possível navegar pela memória investigando os valores de outros endereços sem precisar saber a qual variável eles pertencem.



Fonte: ilustração de Clarissa Brasil

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 a 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 presente 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 ("Endereco: %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. Mas 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 considera-o com 4 bytes. Como *ptr* é um ponteiro do tipo inteiro, precisamos nos certificar de que o retorno de *malloc* seja um valor compatível, 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
```



REFLITA

Um ponteiro aponta para um endereço na memória, seja ele de uma variável pré-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, isso 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.

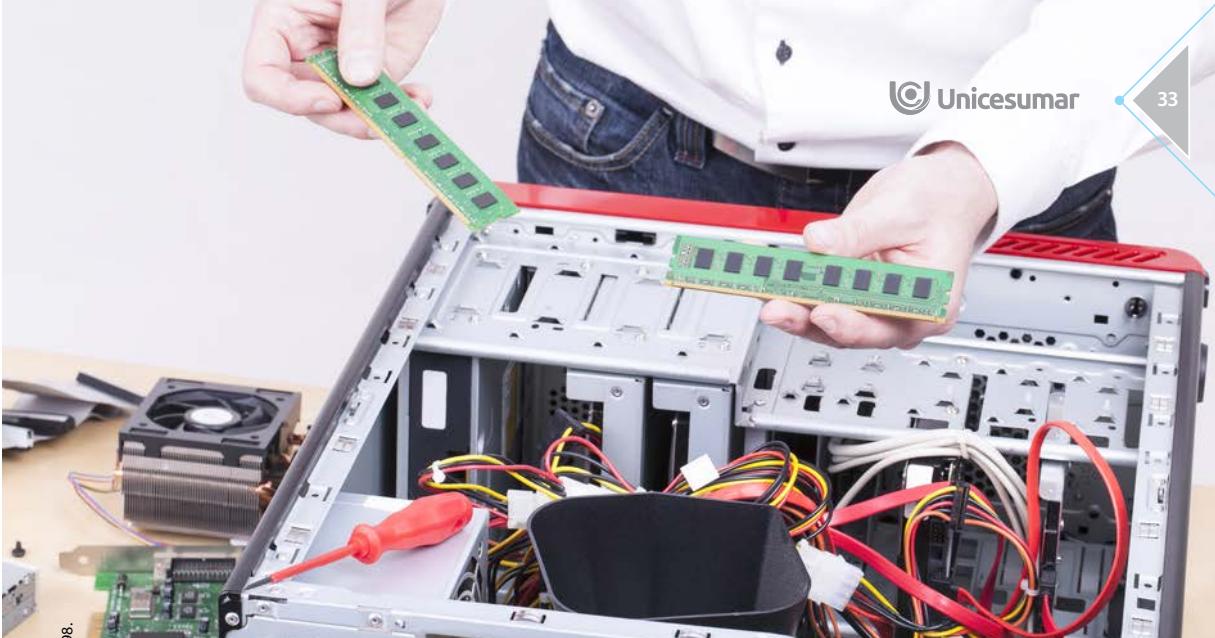
```
Endereco: 00000008

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. Independente 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 àquela 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 a 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 programinha 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 essa:

```
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);  
}
```

CONSIDERAÇÕES FINAIS

Até o presente momento você programou de forma estática. Ficou limitado(a) 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 nesta unidade. O ponteiro é uma estrutura de dados muito poderosa e permite, além de outras coisas, fazer alocações dinâmicas na memória.

Na Unidade 3 experimentaremos um pouco mais do poder dos ponteiros na criação de listas encadeadas, duplamente encadeadas e listas circulares, em que o limite será o tamanho da memória do computador.



ATIVIDADES



1. Faça um pequeno programa para testar seus conhecimentos sobre ponteiros e alocação dinâmica na memória.
 - a) Defina um ponteiro do tipo inteiro.
 - b) Faça alocação dinâmica para o ponteiro recém-criado.
 - c) Preencha a memória com o valor 42.
 - d) Imprima o endereço do ponteiro na memória e o valor contido nele.
2. Uma famosa fábrica de semáforos está testando um novo protótipo, menor, mais barato e eficiente. Foi solicitado à equipe de TI um programa em linguagem C para fazer o teste do novo hardware. O semáforo tem três objetos, cada um contém um atributo cor e outro id. Devido às limitações de memória e processamento, é necessária a criação de um ponteiro que vai percorrendo a memória e imprimindo o valor na tela.
 - a) Crie uma estrutura que tenha dois atributos: cor (cadeia de caracteres) e id (inteiro).
 - b) Crie três variáveis com o tipo definido no item a.
 - c) Crie um ponteiro do mesmo tipo.
 - d) Inicialize as estruturas da seguinte forma:
 - cor vermelha, id 1.
 - cor amarela, id 2.
 - cor verde, id 3.
 - e) Inicialize o ponteiro apontando para a primeira variável criada.
 - f) Por meio de operações aritméticas com ponteiros, percorra a memória e imprima o valor de cada uma das variáveis criadas nesse programa.

ATIVIDADES



3. Qual a diferença entre uma variável do tipo inteira de um ponteiro do tipo inteiro?
4. Por que devemos preencher um ponteiro apenas com o endereço de uma variável ou por alocação dinâmica usando funções como *malloc*?
5. 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.
 - a) Crie uma variável inteira e dois ponteiros do tipo inteiro.
 - b) Pergunte ao usuário o tamanho do vetor dinâmico e leia esse valor na variável inteira.
 - c) Aloque dinamicamente os dois vetores usando a função *malloc*.
 - d) Faça um laço de repetição para preencher o primeiro vetor com potências de 2.
 - e) Faça um segundo laço de repetição para preencher o outro vetor com potências de 3.
 - f) Não se esqueça de usar a biblioteca *math.h* para poder usar o cálculo de potência (*pow*).

MATERIAL COMPLEMENTAR



LIVRO

Projeto de Algoritmos

Nivio Ziviani

Editora: Cengage Learning

Sinopse: esta obra apresenta os principais algoritmos conhecidos. 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.

Destaques:

- As técnicas de projeto de algoritmos são ensinadas por meio de refinamentos sucessivos até o nível de um programa em Pascal.
- Todo programa Pascal tem um programa C correspondente no apêndice.
- Mais de 163 exercícios propostos, dos quais 80 com soluções; 221 programas em Pascal e 221 programas em C; 164 figuras ilustrativas.



PILHAS E FILAS

UNIDADE



Objetivos de Aprendizagem

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

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Pilhas
- Filas

INTRODUÇÃO

Tendo em mente o conceito de estruturas homogêneas e heterogêneas, levaremos em conta o uso de tais estruturas para servirem de base para a criação de estruturas mais complexas como as filas e pilhas.

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 (2002, p. 138-141).

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

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 estar trabalhando 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 do 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 informática, 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).

Agora é preciso remover um valor. 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 se 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, beirando o trivial. 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 3 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, o mesmo 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.



REFLITA

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, coloquei 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 //Adicionar um elemento no final da Pilha
49 void pilha_entrar() {
50     if (pilha.fim == tamanho) {
51         printf("\nA pilha está cheia, impossível empilhar!\n\n");
52         system("pause");
53     }
54     else {
55         printf("\nDigite o valor a ser empilhado: ");
56         scanf("%d", &pilha.dados[pilha.fim]);
57         pilha.fim++;
58     }
59 }
60 }

61 //Retirar o último elemento da Pilha
62 void pilha_sair() {
63     if (pilha.ini == pilha.fim) {
64         printf("\nA pilha está vazia, impossível desempilhar!\n\n");
65         system("pause");
66     }
67     else {
68         pilha.dados[pilha.fim-1] = 0;
69         pilha.fim--;
70     }
71 }
72 }

73 //Mostrar o conteúdo da Pilha
74 void pilha_mostrar() {
75     int i;
76     printf("[  ");
77     for (i = 0; i < tamanho; i++) {
78         printf("%d  ", pilha.dados[i]);
79     }
80     printf("]\n\n");
81 }
82 }

83 //Mostrar o menu de opções
84 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 }
91 }
```

Linhas 1 a 4

Inclusão de bibliotecas necessárias para o funcionamento do programa.

Reprodução proibida. Art. 184 do Código Penal e Lei 9.610 de 19 de fevereiro de 1998.

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. Para mais detalhes sobre prototipação, consulte o livro de Algoritmos e Lógica de Programação II.

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 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 5 pessoas e que o restante dos apostadores esperam 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 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 a sua esquerda.

Existem outras formas de fila, como, a fila cíclica. 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 a 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?

REFLITA



É comum na literatura o uso do termo *enqueue* (ou *push_back*) para indicar a inserção e *dequeue* (ou *pop*) para a remoção em filas.

Segue agora o código-fonte completo de uma fila em linguagem C. Fiz 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 //Protipaçao
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(){
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 //Mostrar o menu de opções
88 void menu_mostrar() {
89     printf("\nEscolha uma opção:\n");
90     printf("1 - Incluir na Fila\n");
91     printf("2 - Excluir da Fila\n");
92     printf("0 - Sair\n\n");
93 }
94 }
```

Linhas 1 a 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. Para mais detalhes sobre prototipação, consulte o livro de Algoritmos e Lógica de Programação II.

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.

CONSIDERAÇÕES FINAIS

Na unidade anterior, vimos uma pequena revisão sobre os tipos de variáveis e aprendemos como elas são alocadas na memória. Isso é muito importante, porque uma sequência de bits nada mais é do que um monte de números 1 e 0 sem sentido algum. É a estrutura de dados que determina como ela deve ser lida e qual o seu significado.

Agora foi a vez de revisarmos as estruturas homogêneas e heterogêneas. Elas são fundamentais para que possamos estudar estruturas mais complexas, como as pilhas e as filas.

Os vetores e matrizes permitem que tenhamos numa única variável, uma coleção grande de dados agrupados. Com os registros, podemos criar nova tipagem de variáveis e usá-las para definir estruturas capazes de conter informações de diferentes tipos.

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.

Até o momento, trabalhamos com estruturas estáticas. Depois de definido o tamanho da pilha ou da fila, ele permanecerá o mesmo até o final da execução do programa. Na próxima unidade, aprenderemos sobre ponteiros e a sua utilização na criação de estruturas dinâmicas sem tamanho ou limite pré-definido.

ATIVIDADES

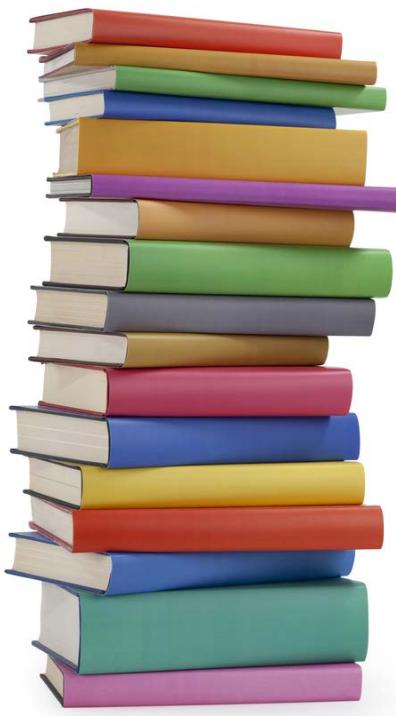


1. Quando um livro é devolvido na Biblioteca do Unicesumar, o funcionário responsável pelo recebimento coloca o livro em cima de uma pilha de livros na mesa ao lado da recepção. O auxiliar de bibliotecário pega o livro do topo da pilha, verifica o seu código e leva-o para o seu devido local no acervo.

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 do mesmo 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.

- 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.
- Defina a variável que será um vetor do tipo pilha de livros.
- Faça uma função de empilhamento, lembrando-se de verificar se a pilha atingiu o tamanho máximo de livros (a mesa não aguenta muito peso).
- Crie uma função para desempilhamento de livros. Não se esqueça de que é necessário verificar se ainda existem livros para ser guardados.
- Elabore uma função que apresente na tela a lista de todos os livros que se encontram empilhados ao lado da recepção.



ATIVIDADES



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 encontrem 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).
 - 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.



MATERIAL COMPLEMENTAR



LIVRO

Estruturas de Dados

Fabiana Lorenzi, Patrícia Noll de Mattos e Tanisi Pereira de Carvalho

Editora: Thomson Learning

Sinopse: na ciéncia da computação 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 desta 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.



NA WEB

O crescimento dos cursos tecnológicos específicos e com curta duração (2 a 3 anos) gerou uma demanda de livros que tratam diretamente o assunto de maneira clara e eficiente. Este livro é indicado aos cursos tecnológicos, para estudantes e profissionais que precisem dominar os conceitos e os algoritmos de forma rápida e precisa:

Web: <http://www.mlaureano.org/livro/livro_estrutura_conta.pdf> (Capítulos 3 e 4).



LISTAS DINÂMICAS

Objetivos de Aprendizagem

- Aprender o conceito de listas.
- Usar a alocação dinâmica de memória na construção de listas.
- Entender como se dá a navegação em listas.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Fundamentos de listas dinâmicas
- Implementando uma lista dinâmica
- Lista dinâmica em forma de pilha
- Lista dinâmica em forma de fila

INTRODUÇÃO

Até agora, usamos os vetores como base para a implementação de estruturas mais complexas como pilhas e listas. Apesar da facilidade e praticidade dos vetores, esse tipo de dado possui diversas limitações.

Veremos agora uma nova estrutura chamada Lista. Apesar de ser parecida com o que vimos até então, 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 nesta unidade a criação e aplicação das listas dinâmicas.

FUNDAMENTOS DE LISTAS DINÂMICAS

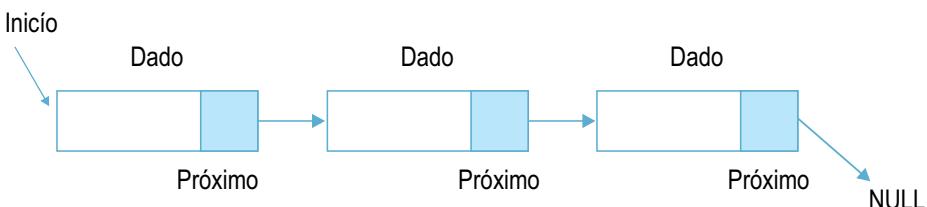
A maneira mais simples de guardar um conjunto de informação na memória se dá pelo uso de vetores. Devido as suas características, o vetor mantém os dados armazenados numa estrutura sequencial. Imaginemos uma variável do tipo vetor de inteiros chamada *vec*. Para acessarmos a quinta posição desse vetor, precisamos apenas usar o índice correto na leitura do vetor *vec[4]*, 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						
Valores	0	1	2	3	4	5
Índices	15	22	50	13	42	77

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.

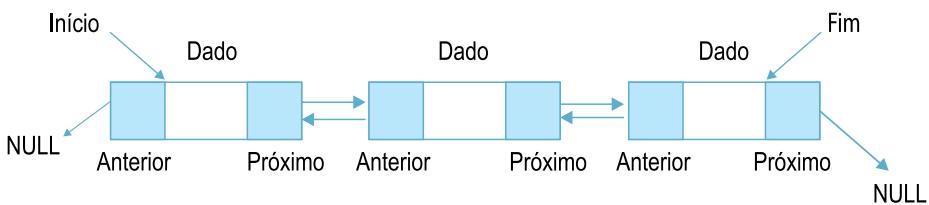
Não é um problema quando se trabalha com poucos dados, mas imagine uma fila com 100, 500, 1000 ou mais posições. Dá para “mover” o início da fila alterando o valor de uma variável auxiliar criada para indicar qual a posição de início da fila no vetor, mas aí o início da fila não seria no início do vetor, 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, p. 224), tal ordenação explícita faz surgir uma estrutura de dados que é conhecida como *lista ligada linear*. Na literatura, também pode-se encontrar referências à tal estrutura como *lista encadeada*.

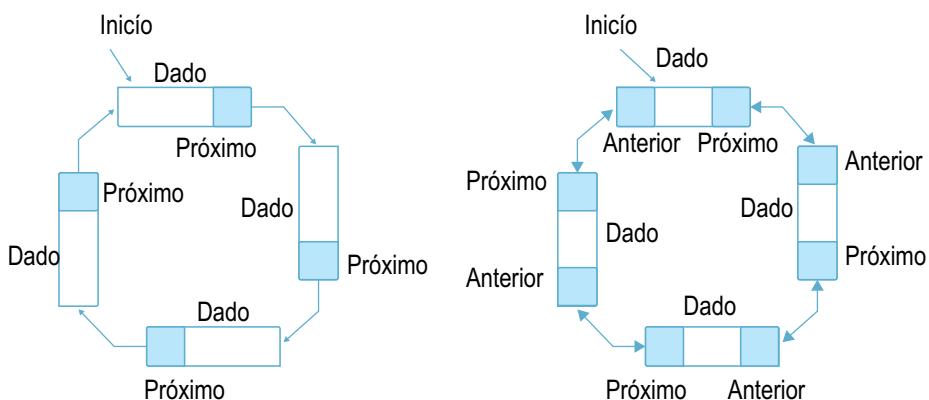


Cada item na lista é chamado de *nó* e contém pelo menos dois elementos: um de *dados* e um de *endereço*. O campo de dados contém o real elemento da lista e o campo de endereço é um ponteiro para o próximo nó. É preciso também uma variável extra que contenha o endereço do primeiro elemento da lista.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

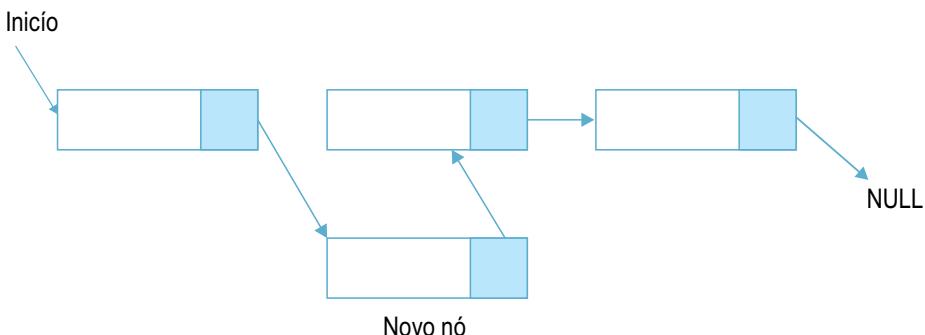


Outro tipo de listas, são as circulares, que são estruturas de dados que “não têm fim”. É como se fosse uma lista ligada linear simples, porém, ao invés de o último elemento apontar para null, ele aponta para o “primeiro” elemento da lista. Seria possível, ainda, combinar o conceito de lista circular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

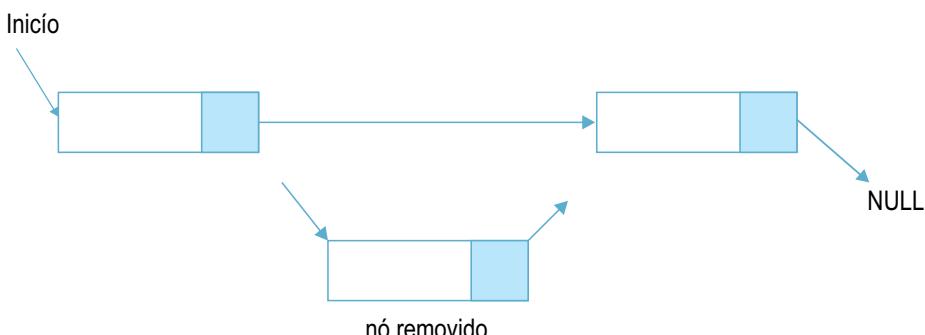


No caso do vetor, todo o espaço é reservado na memória pelo compilador, independendo se estamos usando uma, nenhuma ou todas as suas posições. Já com uma lista dinâmica isso não ocorre, pois vamos adicionando nós sempre que for necessário.

A inserção de um novo elemento numa lista é muito simples. 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 abaixo ilustra bem esse conceito.



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



IMPLEMENTANDO UMA LISTA DINÂMICA

Romperemos agora a barreira da conceituação teórica e vamos nos aventurar no maravilhoso mundo da coisa prática. Não há forma melhor de aprender a programar do que programando.

A primeira coisa de que precisaremos 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 apontará para o próximo nó.

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

Com a estrutura já definida, precisamos criar agora um ponteiro do tipo nó. Ele 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 *proxima* apontando para *null*.

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

Conforme já visto neste livro, a função *malloc* precisa, como parâmetro *sizeof*, o 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 iremos utilizar para desenhar na tela o conteúdo da nossa 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 *proximo* 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.

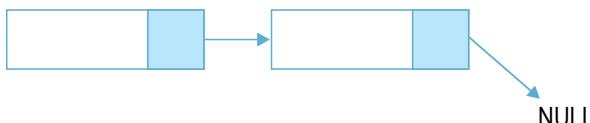
Vamos redesenhar a função *lista_mostrar()*, fazendo com que a verificação da nulidade do atributo *proximo* esteja definida como condicional do laço.

```
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 se faz 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 *proxímo* do último nó.



Criamos um novo nó e fazemos o último nó da lista apontar para ele.



Agora fazemos o novo nó que recém adicionamos na lista aportar para null.



Resumindo: 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;
}
  
```

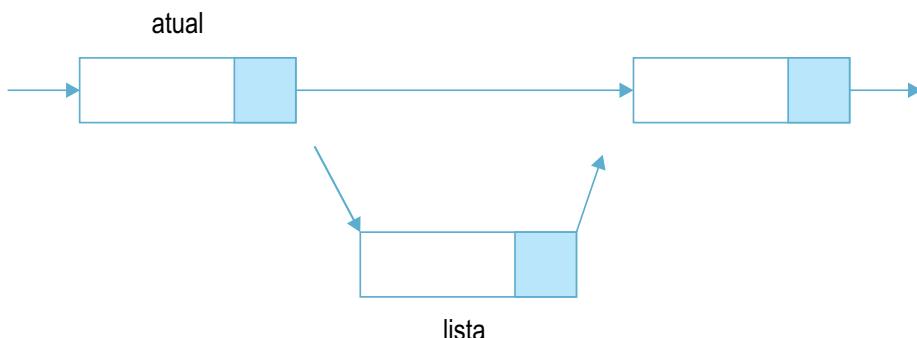
Para diferenciar os nós, toda vez que um novo elemento é adicionado à lista, atribuímos um valor aleatório ao seu atributo *dados*. Isso é possível utilizando a função *rand()* contida na biblioteca *<time.h>*.

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. Vamos criar 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 percorra a lista a partir do primeiro elemento até a posição que queremos remover. Se a estrutura do nó possuisse 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 *lista* estava apontando.



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 agora, 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, permite a inserção de um nó no seu final e a remoção de qualquer um de seus elementos.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 //Definindo a estrutura da lista
6 struct no {
7     int dado;
8     struct no *proximo;
9 };
10
11 //Definindo variáveis
12 typedef no *ptr_no;
13 ptr_no lista;
14 int op;
15
```

```
16 //Prototipação
17 void menu_mostrar();
18 void menu_selecionar(int op);
19 void lista_inserir(ptr_no lista);
20 void lista_remover(ptr_no lista);
21 void lista_mostrar(ptr_no lista);
22
23 //Função Principal
24 int main() {
25     //Inicializando máquina de números randômicos
26     srand(time(NULL));
27     op = 1;
28     //Criando o primeiro nó da lista
29     lista = (ptr_no) malloc(sizeof(no));
30     lista->dado = 0;
31     lista->proxim = NULL;
32     //Laço principal
33     while (op !=0){
34         system("cls");
35         menu_mostrar();
36         scanf("%d", &op);
37         menu_selecionar(op);
38     }
39     system("Pause");
40     return(0);
41 }
42
43 //Mostra o menu de opções
44 void menu_mostrar(){
45     lista_mostrar(lista);
46     printf("\n\nEscolha uma das opcoes:\n");
47     printf("1 - Inserir no final da Lista\n");
48     printf("2 - Remover um item da Lista\n");
49     printf("0 - Sair\n\n");
50 }
51
```

```
52 //Executa a opção escolhida no menu
53 void menu_selecionar(int op){
54     switch (op) {
55         case 1:
56             lista_inserir(lista);
57             break;
58         case 2:
59             lista_remover(lista);
60             break;
61     }
62 }
63
64 //Insere um elemento no final da Lista
65 void lista_inserir(ptr_no lista){
66     while(lista->proximo != NULL) {
67         lista = lista->proximo;
68     }
69     lista->proximo = (ptr_no) malloc(sizeof(no));
70     lista = lista->proximo;
71     lista->dado = rand()%100;
72     lista->proximo = NULL;
73 }
74
75 //Remove um elemento da Lista
76 void lista_remover(ptr_no lista){
77     int dado;
78     ptr_no atual;
79     atual = (ptr_no) malloc(sizeof(no));
80     printf("\n\nEscolha uma dos itens:\n");
81     scanf("%d", &dado);
82     while((lista->dado != dado) ) {
83         if (lista->proximo == NULL) {
84             break;
85         }
86         atual = lista;
87         lista = lista->proximo;
88     }
89     if (lista->dado == dado) {
90         atual->proximo = lista->proximo;
91     }
92 }
```

```
94 //Desenha o conteúdo da Lista na tela
95 void lista_mostrar(ptr_no lista){
96     system("cls");
97     while(1) {
98         printf("%d, ", lista->dado);
99         if (lista->proximo == NULL) {
100             break;
101         }
102         lista = lista->proximo;
103     }
104 }
105
106 //Desenha o conteúdo da Lista na tela
107 void lista_mostrar_2(ptr_no lista){
108     system("cls");
109     while(lista->proximo != NULL) {
110         printf("%d, ", lista->dado);
111         lista = lista->proximo;
112     }
113     printf("%d, ", lista->dado);
114 }
```

LISTA DINÂMICA COM FORMA DE PILHA

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, pode-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, segue 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 opções:\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);
}
```

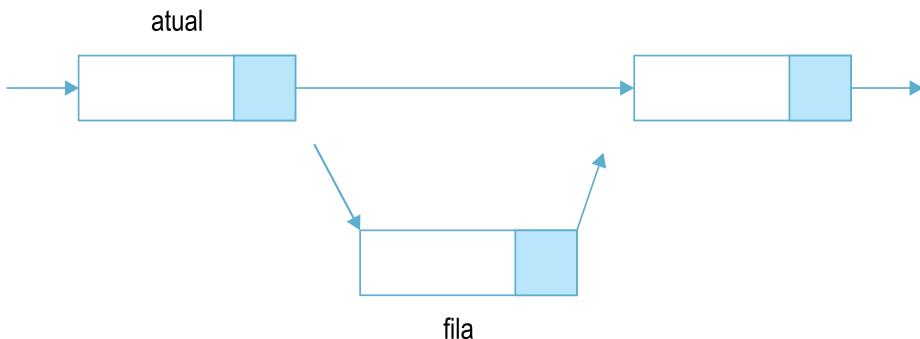
LISTA DINÂMICA COM FORMA DE FILA

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.



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 *fila_remover()* apresentada abaixo também verifica se a fila não está vazia antes de 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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 //Definindo a estrutura da fila
6 struct no {
7     int dado;
8     struct no *proximo;
9 };
10
11 //Definindo variáveis
12 typedef no *ptr_no;
13 ptr_no fila;
14 int op;
```

```
16 //Prototipação
17 void menu_mostrar();
18 void menu_selecionar(int op);
19 void fila_inserir(ptr_no fila);
20 void fila_remover(ptr_no fila);
21 void fila_mostrar(ptr_no fila);
22
23 //Função Principal
24 int main() {
25     //Inicializando máquina de números randômicos
26     srand(time(NULL));
27     op = 1;
28     //Criando o primeiro nó da fila
29     fila = (ptr_no) malloc(sizeof(no));
30     fila-> dado = 0;
31     fila-> proximo = NULL;
32     //Laço principal
33     while (op !=0){
34         system("cls");
35         menu_mostrar();
36         scanf("%d", &op);
37         menu_selecionar(op);
38     }
39     system("Pause");
40     return(0);
41 }
42
43 //Mostra o menu de opções
44 void menu_mostrar(){
45     fila_mostrar(fila);
46     printf("\n\nEscolha uma das opcoes:\n");
47     printf("1 - Inserir no final da fila\n");
48     printf("2 - Remover no inicio da fila\n");
49     printf("0 - Sair\n\n");
50 }
```

```
52 //Executa a opção escolhida no menu
53 void menu_selecionar(int op){
54     switch (op){
55         case 1:
56             fila_inserir(fila);
57             break;
58         case 2:
59             fila_remover(fila);
60             break;
61     }
62 }
63
64 //Insere um elemento no final da fila
65 void fila_inserir(ptr_no fila){
66     while(fila->proximo != NULL){
67         fila = fila->proximo;
68     }
69     fila->proximo = (ptr_no) malloc(sizeof(no));
70     fila = fila->proximo;
71     fila->dado = rand()%100;
72     fila->proximo = NULL;
73 }
74
75 //Remove um elemento do início da fila
76 void fila_remover(ptr_no fila){
77     ptr_no atual;
78     atual = (ptr_no) malloc(sizeof(no));
79     atual = fila;
80     if (fila->proximo != NULL){
81         fila = fila->proximo;
82         atual->proximo = fila->proximo;
83     }
84 }
85
86 //Desenha o conteúdo da fila na tela
87 void fila_mostrar(ptr_no fila){
88     system("cls");
89     while(fila->proximo != NULL) {
90         printf("%d, ", fila->dado);
91         fila = fila->proximo;
92     }
93     printf("%d, ", fila->dado);
94 }
```

CONSIDERAÇÕES FINAIS

Nesta unidade, 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 nesta unidade duas regras:

- 1) Somente será permitido inserir elementos no final da lista.
- 2) 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 permitam navegá-la de forma sequencial mesmo que os dados não estejam fisicamente em sequência.

Na próxima unidade, veremos de perto a Teoria dos Grafos. Mais do que uma simples estrutura de dados, os grafos são formas de modelagem de problemas. Se um problema puder ser modelado na forma de um grafo, é possível aplicar a ele um algoritmo conhecido para encontrar a solução do problema.

ATIVIDADES



1. É possível criar uma lista de forma estática usando um vetor?
2. Crie a estrutura de um nó para uma lista encadeada.
3. Crie a estrutura de um nó para uma lista duplamente encadeada.
4. Qual a vantagem de uma lista duplamente encadeada em relação a uma lista simples?
5. Como sabemos qual é o nó inicial de uma lista simples?
6. Como sabemos qual é o nó final de uma lista simples?
7. 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?

MATERIAL COMPLEMENTAR



LIVRO

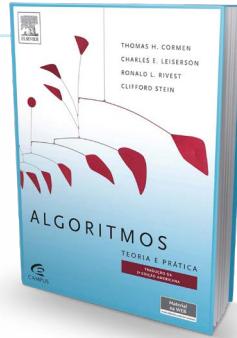
Algoritmos

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Editora: Elsevier

Sinopse: 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.



GRAFOS

UNIDADE

IV

Objetivos de Aprendizagem

- Conhecer a origem dos grafos.
- Aprender suas propriedades.
- Representar grafos computacionalmente.
- Estudar exemplos de aplicação de grafo na modelagem de problemas.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Sete pontes de Königsberg
- Teoria dos Grafos
- Grafos como representação de problemas
- Representação computacional de grafos
- Implementando Grafos em C

INTRODUÇÃO

Agora 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 Tenenbaum (2003, pp. 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.

Quando você busca um endereço pelo Google Maps, ou num GPS, as informações das ruas estão armazenadas em forma de grafo e algoritmos conhecidos são aplicados para apresentar a você possíveis caminhos de onde você está no momento até o destino desejado.

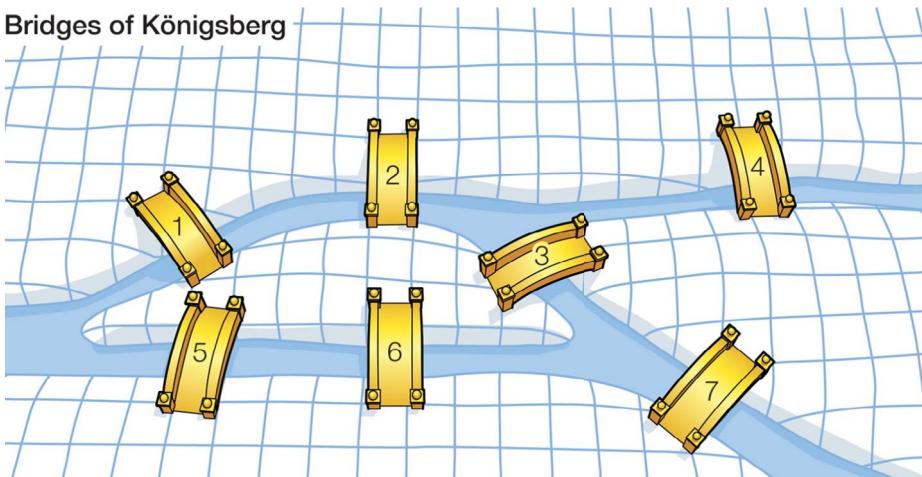
Veremos nesta unidade onde o grafo surgiu e quem o criou. Abordaremos a parte teórica da sua definição, seu conceito e a sua utilização. Ensinaremos também a forma de criar uma representação gráfica a partir de um conjunto de dados e vice-versa. Finalmente, ensinaremos uma das muitas formas de modelagem computacional de grafos.

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.

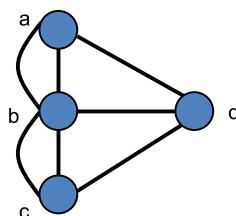
Bridges of Königsberg



Fonte: <<http://corbettmaths.files.wordpress.com/2012/02/bridges-of-konigsberg.jpg>>

Numa 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.

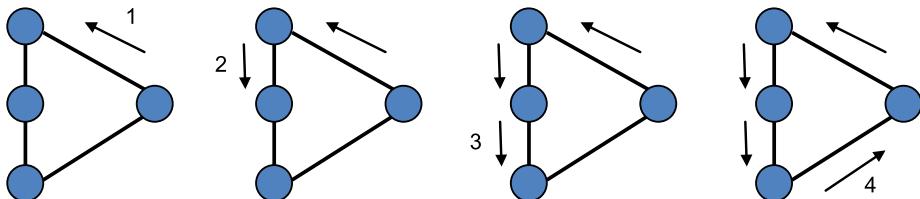
Euler provou matematicamente que não havia uma solução possível de satisfazer tais restrições. Para isso, ele modelou o problema de forma abstrata. Considerou cada porção de terra como um ponto (vértice) e cada ponte como uma reta (aresta) que ligava dois pontos (duas porções de terra). Esse é o registro científico formal mais antigo de um grafo.



Grafo criado por Euler para representar o problema das sete pontes de Königsberg

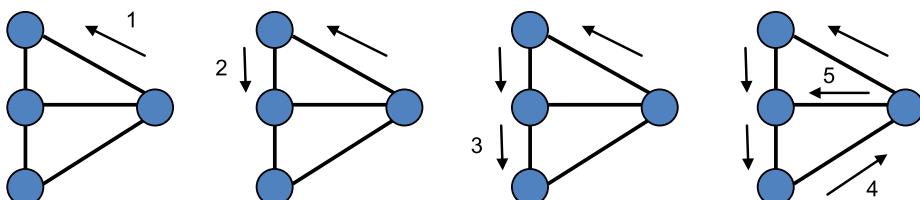
Analisando a figura, ele percebeu que só seria possível caminhar todo o percurso atravessando cada uma das pontes uma única vez se houvesse exatamente zero ou dois vértices de onde saísse um número ímpar de arestas. Falar é fácil, o difícil é provar que isso é uma verdade. Vamos olhar atentamente a figura 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. Isso prova que a primeira afirmação feita por Euler é verdadeira: deve haver exatamente zero vértice com número ímpar de arestas.



Percorrendo um grafo apenas com vértices de número par de arestas

Mas 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.

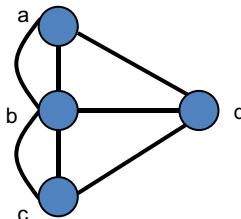


Grafo com exatamente dois vértices com número ímpar de arestas

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 $v \in V$ e $w \in V$. Após a apresentação formal entre você, caro(a) aluno(a), e o dito Grafo, vamos analisar novamente o desenho criado por Euler e traduzir isso para uma linguagem menos matemática.



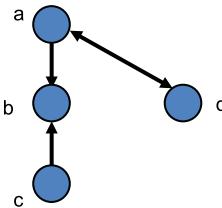
Como acabamos de apresentar, o grafo é formado por dois conjuntos: um de vértices (V , que não pode ser nulo) e um de arestas (E). Dado o grafo de Euler, podemos dizer que o conjunto V é formado por:

$$V = \{a, b, c, d\}$$

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 dos 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*).



Exemplo de grafo orientado

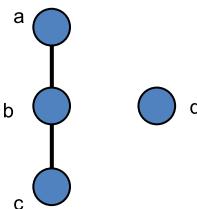
Para indicar que os elementos do conjunto E são pares ordenados, usamos a notação de chaves angulares ao invés de parênteses. A representação do conjunto E do grafo anterior é a seguinte:

$$E = \{(a, b), (c, b), (a, d), (d, a)\}$$

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.



Exemplo de grafo desconexo

Vimos até agora 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

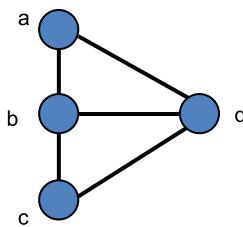
Existem inúmeras formas de representar computacionalmente um grafo, cada qual com suas vantagens e desvantagens em relação a tempo de implementação, uso de memória, gasto de processamento etc. Por questões pedagógicas, vamos apresentar a mais prática.

Sabemos que um grafo é uma estrutura $G = (V, E)$, em que V é um conjunto finito não nulo de *vértices* ou *nós*. 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 $v \in V$ 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:



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.

$$M = \begin{bmatrix} a & b & c & d \\ a & 0 & 0 & 0 \\ b & 0 & 0 & 0 \\ c & 0 & 0 & 0 \\ d & 0 & 0 & 0 \end{bmatrix}$$

Sabemos que existe uma aresta ligando os vértices a e b, então o valor de M_{ab} é 1.

$$M = \begin{bmatrix} a & b & c & d \\ a & 0 & 1 & 0 \\ b & 0 & 0 & 0 \\ c & 0 & 0 & 0 \\ d & 0 & 0 & 0 \end{bmatrix}$$

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.

$$M = \begin{bmatrix} a & b & c & d \\ a & 0 & 1 & 0 \\ b & 1 & 0 & 0 \\ c & 0 & 0 & 0 \\ d & 0 & 0 & 0 \end{bmatrix}$$

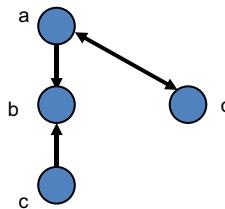
Usando desse raciocínio, vamos completar a matriz M alterando para 1 o valor de cada par ordenado que representar uma aresta entre dois vértices do grafo.

$$M = \begin{bmatrix} a & b & c & d \\ a & 0 & 1 & 0 \\ b & 1 & 0 & 1 \\ c & 0 & 1 & 0 \\ d & 1 & 1 & 0 \end{bmatrix}$$

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.



REFLITA



Quando se trata de um grafo não orientado, podemos dizer que um vértice $a = \{v,w\}$ é igual ao vértice $a = \{w,v\}$, porém num grafo orientado isso não é verdade.

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 0 \\ c & 0 & 1 & 0 & 0 \\ d & 1 & 0 & 0 & 0 \end{bmatrix}$$

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
```

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];
```

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;
}
```

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;
    }
}
```

Precisamos também do efeito oposto, o de 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 que seja necessário transformar o grafo a partir da remoção de arestas. O procedimento será o mesmo, só que ao invés 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;
    }
}
```

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()*, pra 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");
}
```

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");
}
```

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.

```
1 //Bibliotecas
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 //Constantes
6 #define maximo 10
7
8 //Variáveis
9 int tamanho=0;
10 int grafo[maximo];
11 int ma[maximo][maximo];
12 int op=1;
13
14 //Prototipação
15 int grafo_tamanho();
16 void grafo_desenhar();
17 void grafo_desenhar_ma();
18 void grafo_inserir();
19 void grafo_remover();
20 void menu_mostrar();
21
22 //Função Principal
23 int main(){
24     while (tamanho <= 0 || tamanho > maximo) {
25         tamanho = grafo_tamanho();
26         if(tamanho <= 0 || tamanho > maximo) {
27             system("cls");
28             printf("Escolha um valor entre 1 e %d!\n\n", maximo);
29         }
30     else {
31         for(int i=0; i<tamanho;i++) {
32             grafo[i]=i;
33         }
34     }
35 }
36 while (op != 0) {
37     system("cls");
38     grafo_desenhar();
39     grafo_desenhar_ma();
40     menu_mostrar();
41     scanf("%d", &op);
42     switch (op) {
43         case 1:
44             grafo_inserir();
```

```
45     break;
46     case 2:
47         grafo_remover();
48         break;
49     }
50 }
51 system("Pause");
52 return(0);
53 }

54 //Define o número de vértices do Grafo
55 int grafo_tamanho(){
56     int tamanho;
57     printf("Escolha a quantidade de vértices do grafo: ");
58     scanf("%d", &tamanho);
59     return tamanho;
60 }
61 }

62 //Função para desenhar o vetor de vértices
63 void grafo_desenhar(){
64     //Desenhando lista de vértices
65     printf("Listas de vértices\n[  ");
66     for (int i = 0; i < tamanho; i++){
67         printf("%d ", grafo[i]);
68     }
69     printf("]\n\n");
70 }
71 }

72 //Função para desenhar a matriz de arestas
73 void grafo_desenhar_ma(){
74     //Desenhando matriz de adjacências
75     printf("Matriz de adjacencias\n[\n");
76     for (int i = 0; i < tamanho; i++){
77         for (int j = 0; j < tamanho; j++){
78             printf(" %d", ma[i][j]);
79         }
80         printf("\n");
81     }
82     printf("]\n\n");
83 }
84 }

85 //Inserir aresta
86 void grafo_inserir(){
87     int num1, num2;
```

```
89     system("cls");
90     printf("Escolha o vértice de origem entre 0 a %d: ",tamanho-1);
91     scanf("%d",&num1);
92     printf("Escolha o vértice de destino entre 0 a %d: ",tamanho-1);
93     scanf("%d",&num2);
94     if (num1 > tamanho-1 || num2 > tamanho-1 || num1 <0 || num2 <0){
95         printf("\nOs valores precisam estar entre 0 e %d\n\n",tamanho);
96         system("pause");
97     }
98     else {
99         ma[num1][num2]=1;
100        ma[num2][num1]=1;
101    }
102 }
103
104 //Remover aresta
105 void grafo_remover(){
106     int num1, num2;
107     system("cls");
108     printf("Escolha o vértice de origem entre 0 a %d: ",tamanho);
109     scanf("%d", &num1);
110     printf("Escolha o vértice de destino entre 0 a %d: ",tamanho);
111     scanf("%d", &num2);
112     if (num1 > tamanho-1 || num2 > tamanho-1 || num1 <0 || num2 <0){
113         printf("\nOs valores precisam estar entre 0 e %d\n\n",tamanho);
114         system("pause");
115     }
116     else {
117         ma[num1][num2]=0;
118         ma[num2][num1]=0;
119     }
120 }
121
122 //Mostrar o menu de opções
123 void menu_mostrar() {
124     printf("\nEscolha uma opção:\n");
125     printf("1 - Inserir aresta\n");
126     printf("2 - Remover aresta\n");
127     printf("0 - Sair\n\n");
128 }
```

CONSIDERAÇÕES FINAIS

Os grafos são estruturas fascinantes. 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. E se esse algoritmo não existir, você já tem conhecimento suficiente para criá-lo.

ATIVIDADES



Dados os seguintes conjuntos $G = (V,E)$:

- a) $V = \{a\}$, $E = \{\emptyset\}$
- b) $V = \{a,b\}$, $E = \{(a,b)\}$
- c) $V = \{a,b\}$, $E = \{<b,a>\}$
- d) $V = \{a,b,c\}$, $E = \{(a,b), (b,c)\}$
- e) $V = \{a,b,c\}$, $E = \{<a,b>, <c,b>, <a,c>, <c,a>\}$
- f) $V = \{a,b,c,d\}$, $E = \{<a,b>, <d,a>\}$
- g) $V = \{a,b,c,d\}$, $E = \{(a,b), (a,c), (a,d)\}$

1. Desenhe a representação em Grafo de cada conjunto $G = (V,E)$.
2. Classifique cada um dos conjuntos $G = (V,E)$ como sendo orientado, não orientado, conexo e desconexo.
3. Crie a matriz de adjacência para cada conjunto $G = (V,E)$.

MATERIAL COMPLEMENTAR



NA WEB

Algoritmo de Dijkstra para cálculo do Caminho de Custo Mínimo.

Web: <<http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html>>.



NA WEB

Artigos interessantes em outro idioma

Tese de PhD de Edsger Wybe Dijkstra, Communication with an automatic computer, 1959 (inglês).
Web: <<http://www.cs.utexas.edu/users/EWD/PhDthesis/PhDthesis.PDF>>.

Cópia em formato digital da publicação de Euler intitulada Solutio problematis ad geometriam situs pertinentis, 1736 (latim).

Web: <<http://www.math.dartmouth.edu/~euler/docs/originals/E053.pdf>>.

BUSCA EM GRAFOS

UNIDADE



Objetivos de Aprendizagem

- Conhecer os principais algoritmos de busca em grafos e a sua aplicação.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Busca em grafos
- Busca em profundidade
- Busca em largura
- Algoritmo de Dijkstra

INTRODUÇÃO

Essa unidade dá o pontapé inicial no estudo de estruturas de dados mais avançadas. Veremos aqui os dois principais algoritmos de busca em grafos que são a base para técnicas mais complexas e eficientes.

Veremos de perto o funcionamento do algoritmo de Dijkstra, que é uma das melhores soluções de busca de caminhos de menor custo em grafos.

Como leitura complementar, foi inserido um artigo muito interessante dos professores Figueiredo e Gonzaga. Eles mostram um problema real de torres de transmissão, como o modelaram em um grafo e os algoritmos que usaram para procurar a melhor solução.

O artigo é bem extenso e envolve alguns conceitos matemáticos que podem não ser de primeira necessidade para o aluno. Por isso, deixei só as partes pertinentes ao nosso conteúdo. No final, há o link para o artigo completo àqueles que quiserem se aprofundar no assunto.

Lembre-se que se um problema puder ser modelado em um grafo, existem diversos algoritmos prontos que podem ser utilizados ou adaptados para solucioná-lo.

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 específico.

Nesta unidade, 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: a inclusão das bibliotecas *stdlib.h*, *stdio.h* e *stdbool.h* para auxiliar o desenvolvimento; 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; e a declaração do vetor de structs *grafo[MAXV]* que é o grafo em si. Assim, antes de partirmos para os códigos-fonte dos Programas 5.2 e 5.3, é preciso considerar tais pré-requisitos, os quais podem ser visualizados no código-fonte parcial do Programa 5.1 que se segue.

```
1 //Bibliotecas
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stdbool.h>
5
6 //Número máximo de vértices
7 #define MAXV 8
8
9 //Estrutura de um nó
10 typedef struct str_no {
11     int id;
12     struct str_no *proximo;
13 } str_no;
14
15 //Grafo
16 struct str_no grafo[MAXV];
```

Programa 5.1 - Inclusões de biblioteca e declarações globais para implementação das buscas em profundidade e largura

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 5.2 mostra um algoritmo em linguagem C capaz de executar a técnica de busca em profundidade.

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 *buscaEmProfundidade* recebe três parâmetros. O primeiro, *g*, é a lista de adjacências que representa o grafo. O segundo parâmetro é o *inicio*, 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 programa 5.1, que indica o número máximo de vértices que nosso grafo pode representar.

```
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        }
37        if(achou){
38            //Atualizando o nó corrente.
39            corrente = ptr->id;
40        }
41        else{
42            //Não há vértices adjacentes não visitados.
43            //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 }
```

Programa 5.2 - Função *buscaEmProfundidade*.

Fonte: o autor

Inicialmente, durante a execução do Programa 5.2, temos a pilha vazia e, então, empilhamos o vértice inicial de partida da pesquisa. Temos uma variável *indice*, que é usada para controlar o fluxo na pilha. O laço de repetição da linha 14 se repete enquanto houverem 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 *buscaEmProfundidade* 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.

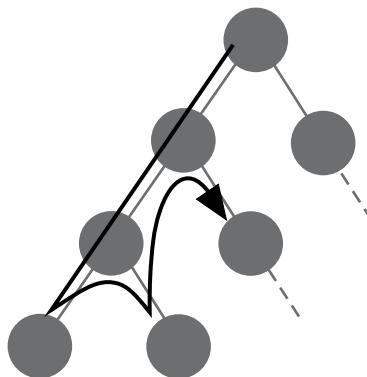


Figura 1 - Exemplo de busca em profundidade

Fonte: Wikimedia ([2018], on-line)¹.

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 ao invés 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.

Observe a Figura 2. 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.

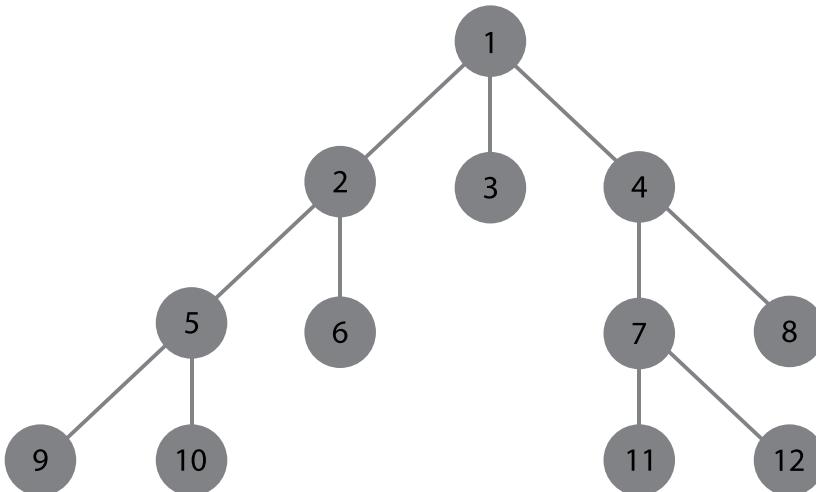


Figura 2 - Ordem que os nós são pesquisados na busca em largura

Fonte: Wikimedia ([2018], on-line)¹.

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, 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 Programa 5.3 traz a função *buscaEmLargura*, implementada em linguagem C. Essa função recebe os mesmos parâmetros da função *buscaEmProfundidade*. Seu funcionamento é praticamente o mesmo que o do Programa 5.2, com a diferença de que empregamos uma *lista* para guiar a ordem de visitação dos nós, ao invés de uma pilha.

```
1 void buscaEmLargura(struct str_no g[], int inicio, int alvo){  
2  
3     int fila[MAXV]; //fila  
4     bool visitado[MAXV]; //nós visitados  
5     int indice = 0; //controle da fila  
6     bool achou = false; //flag (não visitados)  
7     int corrente = inicio;  
8     struct str_no *ptr;  
9     int i;  
10    printf("===== Busca em Largura ===== \n");  
11    //Marcando os nós como 'não visitados'.  
12    for(i=0; i < MAXV; i++)  
13        visitado[i] = false;  
14    //Partindo do primeiro vértice.  
15    printf("VISITANDO: %d. \n", corrente);  
16    visitado[corrente] = true;  
17    fila[indice] = corrente;  
18    indice++;  
19    while(true){  
20        //Visitar os nós adjacentes ao vértice corrente  
21        for(ptr = g[corrente].proximo; ptr != NULL; ptr =  
22            ptr->proximo){  
23            //Caso corrente ainda não tenha sido visitado:  
24            corrente = ptr->id;  
25            if(!visitado[corrente]){  
26                //Enfileira e marca como visitado.  
27                printf("VISITANDO: %d. \n", corrente);  
28                if(corrente == alvo)  
29                {  
30                    printf("Alvo encontrado!\n\n\n");  
31                    return;  
32                }  
33                visitado[corrente] = true;  
34                fila[indice] = corrente;  
35                indice++;  
36            }  
37            //Caso a fila não esteja vazia:  
38            if(indice!=0)  
39            {
```

```

40             //Atualizando vértice corrente.
41             corrente = fila[0];
42             //Desenfileirando o primeiro vértice.
43             for(i=1;i<indice+1;i++)
44                 fila[i-1]=fila[i];
45             indice--;
46         }
47     else
48     {
49         //Não há mais vértices para visitar.
50         printf("Encerrando busca.\n");
51         break;
52     }
53 }
54 return;
55 }
```

Programa 5.3 - Função buscaEmLargura

Fonte: o autor



REFLITA

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 vértice de menor custo.

O Programa 5.4 traz a implementação completa do algoritmo de Dijkstra. Ele tem um menu de opções, permite a criação de um grafo e do 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 finalizado...\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)*vertices*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 Dijasktra\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("? ");
}
```

Programa 5.4 – Algoritmo de Dijkstra em C

Eu separei para você a parte principal do algoritmo de Dijkstra no Programa 5.5. 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;
        }
    }
}
```

Programa 5.5 – O coração de Dijkstra

E 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, ou 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.

CONSIDERAÇÕES FINAIS

Nesta unidade, 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.

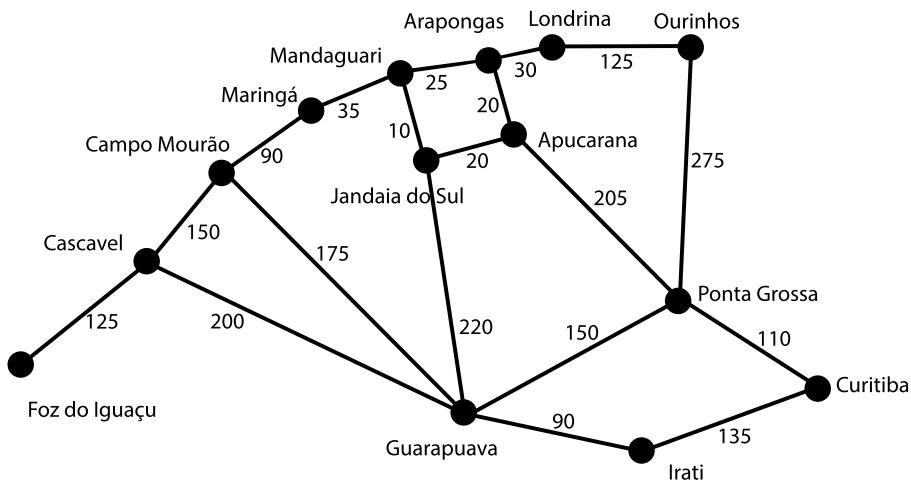
O próprio algoritmo de Dijkstra, que tivemos a oportunidade de estudar, é um algoritmo de busca em largura modificado para uma solução específica, que é encontrar o menor caminho entre dois vértices para grafos que possuam pesos nas arestas.

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, como você verá no artigo **Aplicação de métodos de busca em grafos com nós parcialmente ordenados à locação de torres de transmissão**, apresentado no final desta unidade.

ATIVIDADES



1. Qual é a principal diferença entre a Busca em Largura e a Busca em Profundidade?
2. Em que caso a Busca em Largura é mais eficiente que a Busca em Profundidade?
3. O que diferencia o algoritmo de Dijkstra e a busca em largura?
4. Digite no seu computador o Programa 5.4. Entre com o Grafo (ou parte dele) a seguir e faça simulações de busca de caminho mais curto entre duas cidades.



Fonte: Professeurs Polyml ([2018], on-line)³.



Separei um artigo muito interessante para você. É um trabalho que mostra na prática como a estrutura de grafos e algoritmos de busca são aplicados para a solução de problemas reais. Como o artigo é muito extenso e o seu conteúdo um pouco denso, resolvi fazer alguns recortes, deixando os pontos mais importantes em relação à nossa matéria. Convido você a acessar o link no final do artigo para fazer uma leitura completa e se aprofundar ainda mais sobre o assunto.

APLICAÇÃO DE MÉTODOS DE BUSCA EM GRAFOS COM NÓS PARCIALMENTE ORDENADOS À LOCAÇÃO DE TORRES DE TRANSMISSÃO

Por: João Neiva de Figueiredo; Clóvis C. Gonzaga

Este artigo aborda o problema de locação ótima de torres de transmissão como uma aplicação de métodos de busca em grafos com nós parcialmente ordenados com uma modelagem que aplica a esse problema, pela primeira vez, o conceito de relações de preferência entre nós. São primeiramente apresentados resultados sobre grafos e algoritmos de busca. As restrições eletromecânicas e topográficas à obtenção do caminho de custo mínimo são descritas, são definidos os nós, arcos, custos e caminhos, além de outros componentes do grafo e são descritos os algoritmos de otimização utilizados. O trabalho introduz e demonstra a validade de relações de preferência entre nós, que são utilizadas (juntamente com comparações de custos) no processo de eliminação de caminhos. Este procedimento aumenta a eficiência dos algoritmos de otimização utilizados.

Introdução

Este artigo apresenta algoritmos para otimização da locação de torres de transmissão e baseia-se numa modelagem inovadora do problema de locação de torres como um problema de decisões seqüenciais com a construção recorrente de um grafo através da aplicação repetida de um operador sucessor. O trabalho introduz um processo de eliminação de caminhos no grafo baseado em relações de preferência entre nós que não havia sido utilizado anteriormente para resolver este tipo de problema, e que resulta em aumento de eficiência na determinação de uma linha de transmissão de custo mínimo. O objetivo do trabalho é apresentar uma modelagem original para o problema de locação de torres de transmissão que apresenta melhorias sobre algoritmos consagrados por incorporar a estes relações de preferência entre nós. Acreditamos que esta forma de modelagem tem amplas aplicações em outros tipos de problemas.



Descrição do problema

O projeto de uma linha de transmissão compreende diversas etapas: especificação das características elétricas da linha, escolha do traçado, determinação de características mecânicas das torres, levantamento topográfico e, finalmente, a locação das torres de transmissão sobre o traçado escolhido. Da etapa de determinação de características mecânicas das torres resulta a definição dos tipos de torres que serão utilizadas na linha. Resulta também desta etapa a especificação detalhada de todos os limites mecânicos que cada tipo de torre pode suportar, as diversas alturas disponíveis para cada tipo de torre e o custo de cada par (tipo, altura) de torre. As restrições mecânicas são numerosas e incluem esforço transversal máximo, esforço vertical mínimo para os cabos de transmissão e para os cabos pára-raios, trações longitudinais e transversais para cadeias em torres de suspensão bem como outras restrições particulares a torres de ancoragem e de suspensão. Os custos de cada torre dependem apenas de seu tipo e altura.

A etapa seguinte é a especificação das restrições topográficas com o completo levantamento do perfil do terreno ao longo do traçado da linha. As restrições topográficas incluem a altura de segurança do cabo (altura mínima do cabo ao solo), a topografia do terreno, trechos de locação proibida (como, por exemplo, rios e estradas a serem cruzados pela linha) e pontos de locação obrigatória. O cabo suspenso entre duas torres toma o formato de uma catenária, que neste trabalho (e normalmente) é aproximada por uma parábola. Ela deve necessariamente superar uma distância mínima do solo em todos os pontos do perfil topográfico (a altura de segurança). A catenária de segurança é a catenária virtual que tangencia o perfil topográfico, partindo do ponto na torre que representa sua altura decrescida da altura de segurança naquele ponto do traçado. Neste trabalho, o termo catenária é utilizado para significar catenárias de segurança, as ilustrações mostram torres com catenárias de segurança e as torres têm suas alturas decrescidas da altura de segurança. Ao final desta etapa são conhecidas as restrições mecânicas e as restrições topográficas para o problema e têm-se todos os dados para iniciar o processo de otimização – os tipos e alturas (e consequentemente custos) das torres de transmissão, bem como a descrição completa do perfil topográfico do terreno.

Os esforços mecânicos a que está submetida uma torre somente são determinados quando são conhecidas as localizações e alturas das duas torres adjacentes a ela, ou seja, o tipo e consequentemente o custo de uma torre depende de sua localização e também da localização e altura da torre que a precede e da torre que a segue no traçado. Consequentemente o procedimento de locação de torres passa pelo exame de trechos de perfil topográfico imediatamente à frente da última torre locada, para identificar a melhor opção para a locação da próxima torre e determinar o tipo e o custo da torre anterior. Como será descrito abaixo, modelamos este problema através da utilização de um grafo que incorpora todas as opções possíveis de locação para a linha. Este grafo é criado recursivamente através de um operador sucessor aplicado de forma seqüencial às torres, que compõem o extremo de linha de transmissão mais promissor a cada momento. Após um breve resumo da literatura, será descrita a modelagem bem como os algoritmos para solução do problema.





O algoritmo de Dijkstra

O algoritmo de Dijkstra pode ser aplicado diretamente ao problema. Devido ao fato de o problema ser de natureza contínua, dificilmente dois caminhos terão o *mesmo* nó terminal. Conseqüentemente haverá poucas eliminações de caminhos, e as listas podem crescer explosivamente. Mesmo com a utilização de tolerâncias para comparações entre nós, o número de eliminações pode ser insuficiente. O melhor método para reduzir as listas consiste na utilização de uma relação de preferência entre nós, descrita abaixo.

O algoritmo A*

O algoritmo A*, que reduz o número de expansões feitas por Dijkstra, depende de uma subestimativa para o custo de uma linha de transmissão entre um nó dado e o fim do perfil topográfico. Esta subestimativa pode ser facilmente obtida através do cálculo do custo de uma linha remanescente que tenha mesmo comprimento em um “perfil topográfico perfeito” com ondulações acompanhando as catenárias entre as torres.

Conclusão

Este trabalho apresentou um algoritmo para otimização de locação de torres de transmissão que propõe a otimização global de todo o traçado e utiliza algoritmos eficientes de busca em grafos, com a incorporação de uma relação de preferência entre nós além de comparações de custo. Este procedimento resulta em um aumento de eficiência na determinação de uma linha de transmissão de custo mínimo. A modelagem em teoria de grafos utilizada foi detalhada, foram descritos os algoritmos que se beneficiam da incorporação de relações de preferências entre nós, e foi descrita a relação de preferência entre nós com demonstração de sua validade.

Fonte: adaptado de Figueiredo e Gonzaga (2003).

REFERÊNCIAS

CORMEN, T. H. et al. **Algoritmos**. Tradução de Arlete Simille Marques. 3. ed. Rio de Janeiro: Elsevier, 2012.

FIGUEIREDO, J. N.; GONZAGA, C. C. **Aplicação de métodos de busca em grafos com nós parcialmente ordenados à locação de torres de transmissão**. Rio de Janeiro, v. 23, n. 1, Jan. 2003. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382003000100015&lng=en&nrm=iso>. Acesso em: 17 dez. 2018.

IFRAH, G. **Os números**: história de uma grande invenção. Tradução de Stella Maria de Freitas Senra. 11. ed. São Paulo: Globo, 2005.

MACHADO, Francis Berenger; MAIA, Luiz Paulo. **Arquitetura de Sistemas Operacionais**. 3.ed. Rio de Janeiro: LTC Editora, 2002.

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

TENENBAUM, A. S. **Redes de computadores**. Tradução de Vandenberg D. de Souza. Rio de Janeiro: Elsevier, 2003.

VILELA JUNIOR, A. **Fundamentos e arquitetura de computadores**. Maringá: CESU-MAR, 2012.

REFERÊNCIAS ON-LINE

¹Em: <<http://upload.wikimedia.org/wikipedia/commons/thumb/2/2c/Depthfirst.png/200px-Depthfirst.png>>. Acesso em: 17 dez. 2018.

²Em: <<http://upload.wikimedia.org/wikipedia/commons/3/33/Breadth-first-tree.svg>>. Acesso em: 17 dez. 2018.

³Em: <<http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/IA/Resol-Prob/MapaParana.gif>>. Acesso em: 17 dez. 2018.



GABARITO

UNIDADE I

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>

struct semafaro {
    char cor[10];
    int id;
};

struct semafaro s1 = {"Vermelho", 1};
struct semafaro s2 = {"Amarelo", 2};
struct semafaro s3 = {"Verde", 3};
struct semafaro *ptr_s;

int main(){
    ptr_s = &s1;
    printf("%d - %s\n", (*ptr_s).id, (*ptr_s).cor);
    ptr_s = ptr_s + 1;
    printf("%d - %s\n", (*ptr_s).id, (*ptr_s).cor);
    ptr_s = ptr_s + 1;
    printf("%d - %s\n", (*ptr_s).id, (*ptr_s).cor);
    system("Pause");
    return(0);
}
```



GABARITO

3. Uma variável do tipo inteira aponta pra um número inteiro na memória, já um ponteiro do tipo inteiro aponta para o endereço de um inteiro na memória, seja o de uma variável estática ou dinâmica.
4. Porque se tentarmos alocar indiscriminadamente um endereço qualquer a um ponteiro, corremos o risco de estar manipulando uma área da memória que está sendo utilizada por outro programa ou até mesmo pelo sistema operacional, o que pode causar instabilidade no computador.

5.

```
#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

UNIDADE II

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;
};
```

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 desempilhar!\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;
```

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--;
    }
}
```

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");
}
```



GABARITO

UNIDADE III

1. Sim, a lista pode ser implementada de forma estática num vetor ou dinamicamente na memória. O que caracteriza a lista dinâmica é o fato de seus nós possuírem o endereço do próximo elemento, permitindo assim a sua organização independente do índice do vetor.

2.

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

3.

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

4. Na lista simples, os nós possuem apenas o endereço do próximo elemento. Desse forma, podemos navegar do primeiro ao último nó em um único sentido. Na lista duplamente encadeada, os nós possuem o endereço do elemento anterior e do elemento posterior. Dessa forma, podemos fazer a navegação em ambos os sentidos, tanto do início para o final como do final para o início.

5. Só é possível saber onde começa uma lista dinâmica se tivermos uma variável do tipo ponteiro que aponte para o início da lista.

6. Sabemos se um nó é o último de uma lista se o atributo que aponta para o próximo elemento estiver com o valor nulo (null).

7. 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.



GABARITO

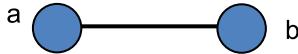
UNIDADE IV

1.

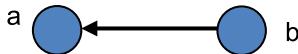
a) $V=\{a\}, E=\{\emptyset\}$



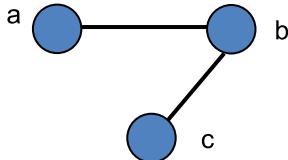
b) $V=\{a,b\}, E=\{(a,b)\}$



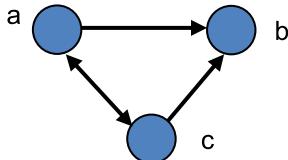
c) $V=\{a,b\}, E=\{(b,a)\}$



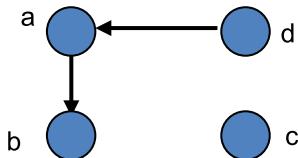
d) $V=\{a,b,c\}, E=\{(a,b), (b,c)\}$



e) $V=\{a,b,c\}, E=\{(a,b), (c,b), (a,c), (c,a)\}$

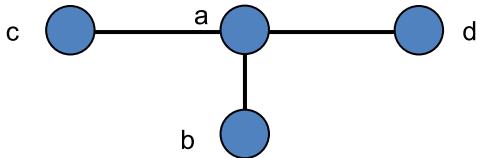


f) $V=\{a,b,c,d\}, E=\{(a,b), (d,a)\}$



GABARITO

g) $V=\{a,b,c,d\}$, $E=\{(a,b), (a,c), (a,d)\}$



2.

Um grafo é dito grafo conexo quando é possível partir de um vértice v até um vértice w através de suas arestas incidentes. Caso contrário, o grafo é dito desconexo.

No caso dos pares de vértices serem ordenados, ou seja, uma aresta $a = \langle v, w \rangle$ é considerada diferente da aresta $a = \langle w, v \rangle$, o grafo é dito orientado (ou dígrafo). Caso contrário, o grafo é não orientado.

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) $V=\{a,b\}$, $E=\{\langle b,a \rangle\}$

Conexo, orientado.

d) $V=\{a,b,c\}$, $E=\{(a,b), (b,c)\}$

Conexo, não orientado.

e) $V=\{a,b,c\}$, $E=\{\langle a,b \rangle, \langle c,b \rangle, \langle a,c \rangle, \langle c,a \rangle\}$

Conexo, orientado.

f) $V=\{a,b,c,d\}$, $E=\{\langle a,b \rangle, \langle d,a \rangle\}$

Desconexo, orientado.

g) $V=\{a,b,c,d\}$, $E=\{(a,b), (a,c), (a,d)\}$

Conexo, não orientado.



GABARITO

3.

a) $V=\{a\}$, $E=\{\emptyset\}$

$$M = \{\emptyset\}$$

b) $V=\{a,b\}$, $E=\{(a,b)\}$

$$M = \begin{bmatrix} & a & b \\ a & 0 & 1 \\ b & 1 & 0 \end{bmatrix}$$

c) $V=\{a,b\}$, $E=\{<b,a>\}$

$$M = \begin{bmatrix} & a & b \\ a & 0 & 1 \\ b & 0 & 0 \end{bmatrix}$$

d) $V=\{a,b,c\}$, $E=\{(a,b),(b,c)\}$

$$M = \begin{bmatrix} & a & b & c \\ a & 0 & 1 & 0 \\ b & 1 & 0 & 1 \\ c & 0 & 1 & 0 \end{bmatrix}$$

e) $V=\{a,b,c\}$, $E=\{<a,b>,<c,b>,<a,c>,<c,a>\}$

$$M = \begin{bmatrix} & a & b & c \\ a & 0 & 1 & 1 \\ b & 0 & 0 & 0 \\ c & 1 & 1 & 0 \end{bmatrix}$$

f) $V=\{a,b,c,d\}$, $E=\{<a,b>,<d,a>\}$

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 0 & 0 \end{bmatrix}$$



GABARITO

g) $V=\{a,b,c,d\}$, $E=\{(a,b),(a,c),(a,d)\}$

$$M = \begin{bmatrix} & a & b & c & d \\ a & 0 & 1 & 1 & 1 \\ b & 1 & 0 & 0 & 0 \\ c & 1 & 0 & 0 & 0 \\ d & 1 & 0 & 0 & 0 \end{bmatrix}$$

UNIDADE V

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. O algoritmo de Dijkstra leva em consideração uma matriz de valores em que estão armazenados os pesos das arestas. Durante a investigação dos vizinhos de um nó, a técnica prioriza adicionar somente o vértice mais próximo, ou seja, aquele que tem a aresta com o menor custo.



CONCLUSÃO

No início deste livro, você foi convidado a fazer uma revisão sobre variáveis compostas homogêneas e heterogêneas. Com essa informação, pudemos recobrar conceitos como o de vetores, matrizes e registros, os quais são fundamentais para a criação de novas abstrações de estruturas de dados.

A memória, que antes era um local atingível apenas pela referência indireta das variáveis, passou a ser uma ferramenta poderosa e acessível ao programador pelo uso dos ponteiros. Agora você, caro(a) aluno(a), pode investigar o conteúdo ali armazenado de forma indireta, navegando livremente pela vastidão de informações carregadas no computador.

E esse foi só o começo. Foi possível se libertar das amarras estáticas criadas pelos tipos pré-definidos de valores e alçar voos maiores, por meio de estruturas dinâmicas, criando o seu próprio espaço de forma arrojada e independente.

Você aprendeu sobre as Listas, estrutura versátil que permite organizar as informações, definindo a sua própria forma de ordenação sequencial, independentemente da posição física que os dados ocupam na memória.

Aprendeu também os conceitos de Pilhas e Filas, que são tipos de Listas que possuem regras rígidas e bem definidas para a entrada e saída de informação.

Por fim, foi apresentada a Teoria dos Grafos, recurso indispensável para qualquer profissional que queira programar de forma eficiente e que permite resolver computacionalmente problemas complexos do dia a dia. Houve, ainda, o contato com as buscas em grafos, em especial, as buscas em profundidade, em largura e o algoritmo de Dijkstra.

A área de Tecnologia da Informação se destaca por ser um ramo da ciência que está em plena evolução. O conhecimento se renova, novas ferramentas surgem a cada dia, os computadores ficam mais rápidos, menores e mais baratos. Entretanto a base ainda é a mesma, e o conteúdo aprendido neste livro irá certamente acompanhá-lo(a) por toda a sua carreira.



ANOTAÇÕES



ANOTAÇÕES



ANOTAÇÕES



ANOTAÇÕES



ANOTAÇÕES

ANOTAÇÕES

