Algoritmos e Estrutura de Dados

Algoritmos e Estrutura de Dados

Kleber Ricardi Rovai

Dados Internacionais de Catalogação na Publicação (CIP)

Rovai, Kleber Ricardi
R873a Algoritmos e estrutura de dados / Kleber Ricardi Rovai.
– Londrina : Editora e Distribuidora Educacional S.A., 2018.
216 p.

ISBN 978-85-522-0660-6

1. Desenho (Projetos). I. Rovai, Kleber Ricardi. II. Título.

CDD 600

Thamiris Mantovani CRB-8/9491

2018
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: http://www.kroton.com.br/

Sumário

Palavras do autor

Caro aluno,

Nos dias atuais, a tecnologia muda muito rapidamente; novos produtos são criados e lançados no mercado. No mundo, a cada dia que passa, mais informações são criadas e utilizadas para gerar novas informações.

Em um mercado cada vez mais disputado, destaca-se o profissional que busca mais conhecimento, está atualizado com as novas tecnologias e desenvolve inovações. De que adianta ter conhecimento de diversas linguagens de programação, das estruturas de dados e acesso a muitas informações, para não as aplicar no dia a dia ou em novas soluções para problemas antigos, inovando?

Nesta disciplina, você estudará a importância das estruturas de dados, como listas ligadas, pilhas e filas, tabelas de espalhamento e armazenamento associativo para o desenvolvimento de aplicações, e de que forma podem nos ajudar a criar soluções melhores para nossos *softwares*. Você conhecerá e compreenderá as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas.

Na Unidade 1, você conhecerá o funcionamento das listas ligadas, assim como as listas duplamente ligadas, criando soluções por meio de aplicações.

Com a Unidade 2 em estudo, teremos a apresentação de pilhas e filas, muito importante no contexto de estruturas dinâmicas.

As tabelas de espalhamento são temas de estudo da nossa Unidade 3, em que poderemos compreender sua construção e aplicação em programas de computador.

Por fim, na Unidade 4, estudaremos o armazenamento associativo, com o qual é possível armazenar em listas os índices, apontando para outras listas de elementos.

É muito importante que você, aluno, desenvolva um planejamento de estudos e acompanhe regularmente as aulas. Aplique todo o conhecimento adquirido com este livro, para aproveitar cada momento da aula.

Seja bem-vindo ao estudo de Algoritmos e Estrutura de Dados e estruture seu futuro!

Listas Ligadas

Convite ao estudo

Caro aluno, seja bem-vindo ao estudo desta unidade. Você estudará as Listas Ligadas em Algoritmos e a Estrutura de Dados e verá como é fácil e prático trabalhar com essas estruturas.

Mas em que utilizo uma estrutura de dados? A resposta para essa questão é: em todos os sistemas utilizamos as estruturas de dados, muitas vezes sem saber. Nos dias atuais, a utilização de estruturas é a base para desenvolvermos sistemas para qualquer área de atividade ou ramo de negócio.

Ao estudar as listas ligadas nesta Unidade 1, você será capaz de conhecer e compreender as listas ligadas, sua construção e uso adequados, sua aplicação em programas de computador, assim como desenvolver um Programa de Computador para Cálculo de Fatoriais com números resultantes maiores que um inteiro longo, sem sinal, por exemplo.

Você foi contratado como programador júnior por uma empresa de desenvolvimento de sistemas que elabora sistemas para diversos setores do comércio, como farmácias, lojas de roupas, padarias etc. Você precisará aplicar todo o seu conhecimento de estrutura de dados em listas ligadas para solucionar esse desafio, utilizando a exibição de lista, as funções inserir e remover, assim como seu conhecimento de pesquisa nessa lista.

Agora, surgiu a demanda de um cliente de autopeças. Tal empresa possui a matriz e uma filial, e você precisará implementar em seus sistemas uma nova funcionalidade nos relatórios. Será necessário, então, gerar uma listagem de produtos com estoque abaixo do número mínimo informado no cadastro do produto no sistema. Esse relatório precisa

fornecer informações de estoque, tanto da matriz quanto da filial, e você será o responsável por analisar essa demanda e criar uma solução para essa funcionalidade no sistema.

Analisando a demanda da loja de autopeças com esta unidade, tenho certeza de que o estudo será fundamental para a compreensão e o aprendizado de Algoritmos e Estrutura de Dados. Preparado para o desafio? Então, vamos lá!

Seção 1.1

Definição e Elementos de Listas Ligadas

Diálogo aberto

Caro aluno,

Nesta seção, você estudará alguns conceitos importantes para a compreensão e o aprendizado sobre as listas ligadas, assim como sua estrutura e suas aplicações.

Você conhecerá e compreenderá as estruturas de dados dinâmicas essenciais e suas aplicações na solução de problemas. Poderá identificar como a estrutura de uma lista ligada funciona e, assim, desenvolver novas soluções para os mais variados sistemas que utilizam essa estrutura como base fundamental do seu funcionamento, ou criar novos processos automatizados baseados em listas ligadas.

Para iniciarmos a situação proposta, imagine que você foi contratado como programador júnior por uma empresa de tecnologia de sistemas que desenvolve sistemas para diversos setores do comércio, como farmácias, lojas de roupas, padarias etc.

Como primeiro desafio, você precisará analisar a demanda de um cliente, uma empresa de autopeças que possui a matriz e uma filial. Ambas utilizam o sistema no qual você trabalha de forma interligada.

Esse cliente precisa de sua ajuda e conhecimento para criar um relatório com informações do estoque mínimo de cada empresa, utilizando uma lista ligada. Você precisará trabalhar com duas listagens de informações e implementar esse relatório.

É bom estar atento ao funcionamento de uma lista e no que esta pode se enquadrar no sistema em que o seu cliente já possui.

Vamos começar?

Bons estudos!

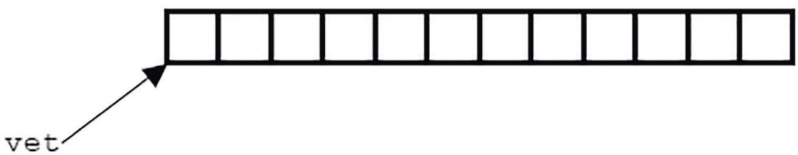
Não pode faltar

As estruturas de dados são formas de organização e distribuição de dados para tornar mais eficientes a busca e manipulação dos dados por algoritmos.

Estudar as estruturas de dados é fundamental para o desenvolvimento de programas e algoritmos, existindo diversos tipos de estrutura de dados para diferentes aplicações específicas em sistemas.

Segundo Celes (2004), a utilização do vetor para representar um conjunto de dados é a forma primitiva de representação, em que podemos definir um tamanho máximo de elementos a ser utilizados nesse vetor. Vejamos o exemplo da Figura 1.1.

Figura 1.1 | Exemplo de vetor



Fonte: Celes et al. (2004, p. 134).

Ainda conforme Celes (2004), o uso do vetor, ao ser declarado, reserva um espaço contíguo na memória para armazenar seus elementos. Assim, é possível acessar qualquer um dos seus elementos a partir do primeiro elemento, por meio de um ponteiro.

Apesar de um vetor ser uma estrutura que permite o acesso aleatório aos elementos, não é uma estrutura flexível de dados, em razão do tamanho máximo que precisa ser definido. Quando temos um aumento de elementos acima do que foi determinado para o vetor, é necessário o incremento da dimensão do vetor para alocá-los. Por exemplo, quando, em uma sala de aula com 25 carteiras disponíveis para alunos, comparecem 30 alunos, sendo necessário alocar mais cadeiras de outras salas para acomodar os alunos excedentes.

Para solucionar esse tipo de problema, é necessário utilizar uma estrutura de dados que permita o crescimento do vetor ou a sua redução de forma dinâmica.

Segundo Celes (2004), uma estrutura de dados que permite esse tipo de armazenamento dinâmico de dados são as listas ligadas, que podemos implementar em diversas outras estruturas de dados.

Definição de listas ligadas

Uma lista ligada é uma coleção L:[a1, a2, ..., an], em que n > 0. Sua propriedade estrutural baseia-se apenas na posição relativa dos elementos, dispostos linearmente.

De acordo com Silva (2007), é também conhecida como lista encadeada. É composta de um conjunto de dados dispostos por uma sequência de nós, em que a relação de sucessão desses elementos é determinada por um ponteiro que indica a posição do próximo elemento, podendo estar ordenado ou não.

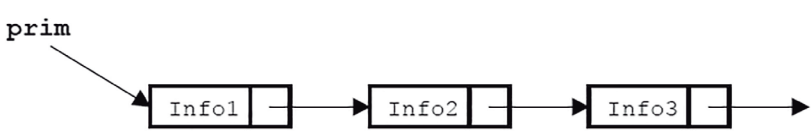
Pesquise mais

É importante o conhecimento de ponteiros para o estudo deste conteúdo e, para isso, é bom relembrar seu conceito e seu funcionamento.

WR KITS. Ponteiros. Linguagem C #020. 2015. Disponível em: <https://www.youtube.com/watch?v=LyvtVTcMIkc>. Acesso em: 11 out. 2017. (Vídeo do YouTube)

Na Figura 1.2, podemos visualizar o modelo de uma lista ligada. Segundo Celes (2004), a estrutura da lista consiste em uma sequência de elementos encadeados, definida como nó de lista. O nó de lista é composto de duas informações: a informação armazenada e um ponteiro que indica o próximo elemento da lista.

Figura 1.2 | Modelo de lista ligada



Fonte: Celes et al. (2004, p. 134).

Diferentemente dos vetores, em que o armazenamento é realizado de forma contígua, a lista ligada estabelece a sequência de forma lógica.

Conforme Silva (2007), para trabalharmos com a lista encadeada, definimos um ponto inicial ou um ponteiro para o começo dela. A partir daí, podemos inserir elementos, remover ou realizar buscas na lista.

As listas apresentam os seguintes procedimentos básicos de manipulação, segundo Silva (2007):

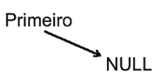
- Criação ou definição da estrutura de uma lista.
- Inicialização da lista.
- Inserção com base em um endereço como referência.
- Alocação de um endereço de nó para inserção na lista.
- Remoção do nó com base em um endereço como referência.
- Deslocamento do nó removido da lista.

Assimile

Uma das vantagens das listas ligadas é que os elementos alocados não precisam estar sequencialmente na memória, como é necessário ocorrer com os vetores. Cada elemento pode estar alocado em diferentes regiões dela.

Quando uma lista está sem nós, é definida como vazia ou nula, assim o valor do ponteiro para o próximo nó da lista é considerado ponteiro nulo, conforme mostra a Figura 1.3.

Figura 1.3 | Lista vazia com ponteiro nulo



Fonte: elaborada pelo autor.

Elementos de dados em listas ligadas

Os elementos de uma lista são armazenados em posições sequenciais de memória, sempre que possível e de forma dinâmica, e permitem que a lista seja percorrida em qualquer direção.

Toda lista precisa ter sua estrutura definida, sabendo que cada nó é composto de um conjunto de informações de tipos diferentes, por um campo de informação e outro de valor inteiro para o ponteiro.

Os elementos de informação de uma lista podem ser do tipo *int, char* e/ou *float*. Ao criar uma estrutura de uma lista, definimos também o tipo de dado que será utilizado em sua implementação. A seguir, veja um modelo de implementação de uma lista:

```
struct lista {
int info;
struct lista* prox;
};
typedef struct lista Lista;
```

Exemplificando

Exemplo de declaração para criar uma lista em C:

```
/*Cria a estrutura da lista*/
struct alunos {
char nome[25];
struct alunos* prox;
};
typedef struct alunos Classe;
```

No exemplo anterior, podemos identificar que:

- será criada uma *struct* (registro) alunos;
- Na *struct*, temos a variável *nome* do tipo *char*, que será nossa informação;
- Temos outra *struct prox* com ponteiro para a própria *struct* alunos, para receber o endereço de apontamento da próxima informação.

Precisamos inicializar a lista para ser utilizada após sua criação. Para isso, basta criarmos uma função em que inicializamos a lista como nula, pois esta é uma das possíveis formas de inicialização:

```
/* Função para inicialização: retorna uma lista vazia */
Lista* inicializa (void)
{
return NULL;
}
```

Elementos de ligação (ponteiros)

Segundo Veloso (1996), os elementos de ligação em uma lista ligada são os ponteiros. Um ponteiro é um tipo de variável que armazena um endereço de memória e não o conteúdo da posição de memória.

A utilização dos ponteiros é indicada nos casos em que é preciso conhecer o endereço que está armazenando a variável. Podemos declarar um ponteiro utilizando a mesma palavra da variável, precedido do caractere * (asterisco). Vejamos o exemplo:

```
int *ptr; /* sendo um ponteiro do tipo inteiro*/
float *ptr; /* sendo um ponteiro do tipo ponto flutuante*/
char *ptr; /* sendo um ponteiro do tipo caracteres*/
```

Exemplificando

```
/* Exemplo de uma estrutura da lista declarada para armazenar dados de uma agenda. */
typedef struct lista {
char *nome;        /*Declaração de um ponteiro do tipo char
int telefone;
struct lista *proximo;
} Dados;
```

De acordo com Tenembaum et al. (2007), um ponteiro é como qualquer outro tipo de variável. Pode ser utilizado de forma dinâmica, para armazenamento e manipulação de valores.

Para sabermos o endereço da memória reservada à variável, utiliza-se o operador & com o nome de uma variável, enquanto o operador *(asterisco), utilizado com a variável do tipo ponteiro, acessa o conteúdo armazenado do endereço de memória, conforme Silva (2007). Temos:

```
int x = 10; /*variável
int *p;    /*ponteiro
p = &x;   /*ponteiro p aponta para o endereço da variável x
```

Em listas, além do uso de ponteiros, utilizamos também as alocações dinâmicas de memória, que são porções de memórias para utilização das listas.

Para compreendermos como funciona um ponteiro em uma lista, precisamos entender a função *malloc( ), Memory Allocation* ou Alocação de Memória. É a responsável pela reserva de espaços na memória principal. Tem como finalidade alocar uma faixa de bytes consecutivos na memória do computador e retornar o endereço dessa faixa ao sistema.

O trecho do código a seguir apresenta um exemplo de utilização da função *malloc( )* e do ponteiro:

```
char *pnt;

pnt = malloc (2); /* Aloca 2 bytes na memória */

scanf ("%c", pnt);
```

O endereço retornado pela função *malloc* é da posição inicial, onde se localizam os *bytes* alocados.

Em uma lista, precisamos alocar o tipo de dado no qual foram declarados seus elementos e, por esse tipo de dados ocupar vários *bytes* na memória, precisaremos utilizar a função *sizeof*, que permite informar quantos *bytes* o tipo de elemento criado terá. Na Figura 1.4, temos um exemplo de código em C, com a utilização das funções *malloc( )* e *sizeof*.

Figura 1.4 | Exemplo de programa em C com *malloc( )* e *sizeof*

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

int main() {

    int *p;
    p=(int *) malloc(sizeof(int));

    if (!p) {
        printf("Erro de memoria insuficiente");
    }else{
        printf("Memoria alocada com sucesso");
    }

    return 0;
}
```

Fonte: elaborada pelo autor.

Pesquise mais

Neste vídeo sobre ponteiros e *malloc*, são apresentados o conceito e a aplicação da alocação de memórias sobre a variável de ponteiro, que será muito importante para a utilização em estruturas de dados do tipo Lista.

G-TECH. Programação em C/C++. Aula 33 – Ponteiros e Malloc. 2014. Disponível em: <https://www.youtube.com/watch?v=rf5BHtVYDIk>. Acesso em: 11 out. 2017. (Vídeo do YouTube)

Podemos classificar as listas de duas formas. Uma delas é a lista com cabeçalho, em que o conteúdo existente no primeiro elemento é irrelevante, pois serve apenas como marcador do seu início. Esse primeiro elemento permanece sempre como ponto inicial na memória, independentemente se a lista está com valores ou não.

Assim, podemos considerar que *start* é o endereço inicial da nossa lista. Para determinar que a lista está vazia, consideramos: *start* ⇒ *prox* == *NULL*.

Como exemplo deste trecho de código, temos:

```
celula *start;
start = malloc (sizeof (celula));
start ⇒ prox = NULL;
```

Exemplificando

Exemplo de um trecho de uma função do tipo inteiro, para inserir pacientes com prioridades na lista:

```
int insere_com_prioridade(Fila *F , Paciente *P){
Lista *ptnodo, *aux , *ant;
ptnodo = (Lista*)malloc(sizeof(Lista));
```

Outra classificação de uma lista é chamada de lista sem cabeçalho, em que o conteúdo do primeiro elemento tem a mesma importância que os demais. Assim, a lista é considerada vazia se o primeiro elemento é NULL. A criação desse tipo de lista vazia pode ser definida por *start* = *NULL*.

Aplicações de listas ligadas

As aplicações de listas no nosso dia a dia são mais comuns do que parece. Podemos aplicar listas em muitas funções. Às vezes, não percebemos, mas estão presentes.

A lista pode ser aplicada em nosso trabalho, quando precisamos listar todas as nossas tarefas do próximo dia de trabalho e, depois, definimos quais delas têm mais prioridade, como na Figura 1.5, e vamos numerando todas as demais por grau de prioridade.

Figura 1.5 | Lista de prioridade de tarefas

Fonte: <http://www.istockphoto.com/br/foto/perto-de-um-espa%C3%A7o-em-branco-lista-de-prioridades-e-caneta-gm532120946-94121105>. Acesso em: 23 out. 2017.

Outra possível aplicação de listas ligadas é a criação de uma lista de compras de supermercado. Ao anotar tudo o que você precisa comprar, automaticamente está criando uma lista na qual podem ser incluídos ou removidos produtos conforme a compra vai sendo realizada. Assim, é possível gerenciar uma lista de produtos de forma dinâmica, como na Figura 1.6, em que há um trecho de código para iniciar uma lista de supermercado.

Figura 1.6 | Trecho de código para iniciar uma lista de supermercado

```
Dados *inicia_listaMerc(char *prod, int numpro) {

    Dados *novo;

    novo = (Dados *)malloc(sizeof(Dados));

    novo -> prod = (char *)malloc(strlen(prod) + 1);
    strncpy (novo -> prod, prod, strlen(prod) + 1);
    novo -> numpro = numpro;
    novo -> proximo = NULL;

    return novo;
}
```

Fonte: elaborada pelo autor.

Podemos também utilizar uma lista para *checklist* de itens para viagem. Você pode criar uma lista e elencar todos os itens de que precisa ou pode levar na viagem que realizará com seus amigos à praia. Após listar todos os produtos que levará nessa viagem, ao separá-los, você pode checar ou remover os que parecem desnecessários e até adicionar novos produtos de que se lembrou posteriormente.

Vamos imaginar que você desenvolverá um sistema para o *site* de uma empresa de casamentos e, por meio desse sistema, os usuários poderão criar a lista de convidados para seu matrimônio, assim como a lista de presentes que os noivos selecionarem. Tanto a lista de convidados quanto a lista de presentes permitem aos noivos adicionarem convidados ou presentes e removê-los da lista, quando necessário. Na Figura 1.7, temos um trecho de um código para adicionar convidados em uma lista:

Figura 1.7 | Exemplo de trecho para inserir convidados em uma lista

```
void convidados inserirConvid(tipoitem elemento,int &cont)
{
    festa *novo,*aux,*aux1;
    aux = inicio;
    aux1 = inicio -> prox;

    while (aux1 != NULL) {
        if(strcmp(aux1 -> item.nome, elemento.nome) > 0)
            break;

        aux = aux -> prox;
        aux1 = aux1 -> prox;
    }

    if((novo = new(festa) == NULL)
        printf("\nMemoria insuficiente");
    else {
        novo -> prox = aux -> prox;
        aux -> prox = novo;
        novo -> item = elemento;
        cont++;
        printf("\nConvidado inserido com sucesso!");
    }

    if (aux1 == NULL)
        fim = novo;

    return;
}
```

Fonte: elaborada pelo autor.

É possível, também, desenvolver um sistema para uma instituição de ensino que permita gerar uma lista de alunos, adicionando ou removendo um aluno da classe, além de agrupá-los por ordem alfabética, mesmo que os RAs ou ordens de matrícula estejam em ordens distintas. Assim, a lista ligada permitiria trazer todos os alunos dessa determinada classe com base no nome de cada aluno.

Com esses exemplos, podemos ver que as aplicações das listas são diversas e estão presentes no nosso dia a dia.

Reflita

Caro aluno, quando utilizamos uma lista ligada para determinada aplicação, em vez de um vetor, permitimos que o nosso sistema trabalhe de forma dinâmica com a memória e a utilização de um ponteiro para indicar qual o próximo elemento da lista. Por que podemos considerar que a lista ligada tem um desempenho melhor do que o uso de um vetor? Teríamos o mesmo resultado utilizando o vetor em vez da lista?

Sem medo de errar

Caro aluno, agora que você já estudou sobre as listas e compreendeu seu funcionamento, retomaremos nossa situação-problema: você foi contratado por uma empresa de desenvolvimento de *softwares* como programador júnior e, como primeiro desafio, precisará analisar a demanda de um cliente, uma empresa de autopeças que possui a matriz e uma filial. Ambas as empresas utilizam o sistema com o qual você trabalha de forma interligada.

Esse cliente precisa de sua ajuda e conhecimento para criar um relatório com informações de estoque mínimo de cada empresa, utilizando uma lista ligada.

Será necessário, então, criar um relatório com informações de estoque mínimo de cada uma. Para isso, você precisará pesquisar como trabalhar com duas listagens de informações e, assim, implementar o relatório.

Ao utilizar a lista ligada, de que maneira as informações dos dois estoques podem ser apresentadas ao cliente em uma única listagem impressa, diretamente na impressora? É possível essa lista de produtos ser impressa em ordem alfabética?

Após o estudo e compreensão das listas ligadas, você pôde entender que, com o uso desse tipo de estrutura de dados, é possível implementar uma solução para o sistema de seu cliente.

É possível criar uma terceira lista ligada, conforme estudamos nesta seção, para receber os produtos dos dois estoques, criando um algoritmo de verificação entre as duas listas. Caso o produto esteja com estoque baixo, o produto vai para essa terceira lista, gerando as informações a serem impressas.

Ao alimentarmos a terceira lista com esses produtos, podemos implementar no algoritmo uma comparação dos produtos pelo nome e alterar a ordem de impressão de cada um, baseando-nos no ponteiro de cada elemento adicionado.

Avançando na prática

Desenvolvendo uma lista de itinerários

Descrição da situação-problema

Uma empresa de transporte coletivo municipal, após estudos realizados com passageiros, identificou que muitas das reclamações eram sobre o ônibus não passar no ponto corretamente.

Com esses dados, a empresa de transporte decidiu desenvolver um sistema para orientar os motoristas sobre a questão e contratou sua empresa para esse desenvolvimento.

Sua empresa precisa, então, desenvolver um sistema em que a empresa de ônibus insira os pontos de parada de cada rota, para que os motoristas sigam esse roteiro e não percam nenhum ponto de passageiros, evitando reclamações dos usuários e desvio da rota.

Com o conhecimento adquirido em listas ligadas, você precisa desenvolver esse sistema para cada rota existente na empresa.

Como é possível criar uma solução para o problema que essa empresa vem enfrentando em seus itinerários? Apresente a solução encontrada à empresa de transportes. Vamos começar?

Resolução da situação-problema

Para resolver o problema enfrentado pela empresa de ônibus, primeiramente precisamos identificar quantas linhas de ônibus estão em operação e saber como funciona o sistema de rotas atual.

Com base nessas informações, você identificará qual é o tipo de estrutura básica e quais variáveis serão criadas no sistema. Ao utilizarmos a criação dos elementos de uma lista ligada, podemos inserir os nós na lista, que seriam os pontos de parada dos ônibus, como uma variável do tipo *char*. Vejamos a seguir:

```
struct rota {
char[15] parada;
struct rota* prox;
};
typedef struct rota Rota;
```

E inicializar a lista ligada com a função:

```
Rota* inicializa (void) {
return NULL;
}
```

Com essa estrutura definida, podemos criar as implementações da função principal *Main* com as funções para inserir uma nova parada no roteiro, excluir uma parada, pesquisar algum ponto de parada, listando todos os pontos de parada de uma rota.

Com os seus conhecimentos em algoritmos, você pode criar as listas responsáveis para cada itinerário, utilizando um controle de repetição para criar um menu no sistema e definir qual rota o usuário deseja alterar, levando em consideração que cada itinerário possui seus pontos específicos e um ponto leva ao ponto seguinte dentro de uma lista ligada.

Faça valer a pena

1. Segundo Silva (2007), uma lista ligada, também conhecida como lista encadeada, é um conjunto de dados dispostos por uma sequência de nós, em que a relação de sucessão desses elementos é determinada por um ponteiro que indica a posição do próximo elemento, podendo estar ordenado ou não.

Assinale a alternativa a seguir que apresenta a informação correta quanto à composição de um nó da lista ligada:

a) Ponteiro para o elemento anterior e uma informação.
b) Uma informação e um ponteiro para o próximo elemento.
c) Ponteiro para o próximo elemento e um ponteiro para o elemento anterior.
d) Ponteiro para o próximo elemento e um dado.
e) Uma informação e um ponteiro para o elemento anterior.

2. Dada a sentença a seguir:

Em uma lista, precisamos _____ o tipo de dado no qual foram declarados os _____ da lista e, por esse tipo de dados ocupar vários *bytes* na _____, precisaremos utilizar a função _____, que nos informa quantos *bytes* o tipo de elemento criado terá.

Com base na sentença, assinale a alternativa que apresenta as palavras que completam a frase corretamente:

a) utilizar, elementos, lista, *sizeof*.
b) alocar, ponteiros, memória, *sizeof*.
c) alocar, elementos, memória, *sizeof*.
d) utilizar, ponteiros, memória, *malloc*.
e) alocar, elementos, lista, *malloc*.

3. Conforme Silva (2007), para trabalharmos com a lista encadeada, definimos um ponto inicial ou um ponteiro para o começo dela. A partir daí, podemos inserir elementos, remover ou realizar buscas nela.

Dadas as afirmativas a seguir sobre procedimentos básicos de manipulação de uma lista:

I. Criação ou definição da estrutura de uma lista.
II. Inserção com base em um endereço como referência.
III. Alocação de um endereço de nó para inserção na lista.
IV. Remoção do nó com base em um elemento apenas.
V. Deslocamento do nó removido da lista.

Assinale a alternativa que contém as afirmativas corretas sobre procedimentos básicos de manipulação:

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

Seção 1.2

Operações com Listas Ligadas

Diálogo aberto

Caro aluno, seja bem-vindo a mais um conteúdo sobre listas ligadas.

Você já se inscreveu em alguma prova de concurso público? Já imaginou a quantidade de pessoas inscritas em um concurso de âmbito nacional? E para gerar uma lista de pessoas inscritas por região? Pois bem, com as listas ligadas, é possível criarmos uma solução para um evento desse porte. Por meio de listas ligadas, podemos ordenar os inscritos por ordem alfabética de nome ou segmentar uma lista por cidades e até mesmo segmentá-la por cidades e, depois, ordená-la por ordem alfabética.

Nesta seção, você estudará conceitos e as aplicabilidades de operações com as listas ligadas, a fim de conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas.

Ao utilizar os conceitos sobre listas ligadas vistos na seção anterior, você estudará e compreenderá, nesta seção, como aplicar as operações para inserir um elemento no início, no meio e no fim da lista ligada, assim como o remover e percorrer uma lista para buscá-lo.

Você conhecerá o funcionamento das operações em uma lista e poderá criar novas funções para aplicá-la em novos sistemas.

Como programador júnior, você precisa aplicar os conhecimentos que está adquirindo em estrutura de dados em listas ligadas, criando os relatórios necessários, para solucionar uma nova demanda de um cliente de autopeças.

Seu cliente solicitou que:

- o relatório seja gerado em tempo real;
- o relatório seja exibido todas as vezes que um produto entrar nessa listagem ou sair dela;
- o relatório seja gerado por meio de venda ou compra de produtos;
- seja possível pesquisar se um produto consta na listagem ou não.

Como desafio, agora você precisa implementar no relatório em que está trabalhando a adição ou remoção de produtos na listagem.

O principal desafio para esta seção é utilizar a exibição de lista, inserir e remover, assim como utilizar seu conhecimento em pesquisar na lista para resolver a solicitação de seu cliente.

Vamos começar?

Não pode faltar

Caro aluno, abordamos na seção anterior alguns conceitos fundamentais e básicos sobre as listas ligadas e seus elementos e aplicações dentro da Estrutura de Dados. Nesta seção, vamos compreender como podemos realizar operações para adicionar, remover, buscar um elemento e percorrer toda a lista ligada.

Uma lista nula (vazia) é uma lista sem nenhum nó, assim o ponteiro para o próximo elemento possui valor nulo, conforme Tenenbaum et al. (2007, p. 225):

> **Uma lista é uma estrutura de dados dinâmica. O número de nós de uma lista pode variar consideravelmente à medida que são inseridos e removidos elementos. A natureza dinâmica de uma lista pode ser comparada à natureza estática de um vetor cujo tamanho permanece constante.**

Adicionar elementos à lista

Ao criarmos uma lista ligada, conforme a estrutura apresentada no *box* Exemplificando, a seguir, nossa lista é criada sem nenhum valor, ou seja, é vazia.

Segundo Celes et al. (2004), para inserirmos um elemento na lista ligada, é necessário alocarmos o espaço na memória, de forma dinâmica, para armazenar o elemento e ligá-lo à lista existente.

Podemos inserir um elemento em uma lista em três situações diferentes. Ao inserirmos uma informação na lista ligada, é imprescindível que seja atualizado o valor do ponteiro dessa lista, assim a lista ligada deve apontar ao novo elemento da lista, segundo Celes (2004).

Inserir um novo elemento no início da lista é a forma mais simples de inserção em uma lista ligada. Podemos implementar a função para adicionar um novo elemento no início da lista, como vemos a seguir:

Exemplificando

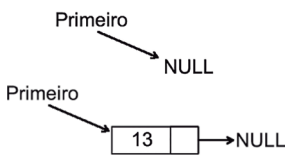
Veja o exemplo de um trecho de implementação do código para inserir um novo elemento no início da lista:

```
Lista* inserir (Lista* l, int i) {
Lista* novo = (Lista*) malloc(sizeof(Lista));
novo -> info = i;
novo -> prox = l;
return novo;
}
```

Podemos visualizar que essa função aloca o espaço, por meio da função *Malloc*, criando um ponteiro para armazenar o novo elemento na lista e a nova informação e apontar para o primeiro elemento da lista.

Na Figura 1.8, temos a lista vazia e, depois, a lista com um elemento inserido, o número 13.

Figura 1.8 | Inserindo um novo elemento no início da lista vazia



Fonte: elaborada pelo autor.

Assim, podemos implementar o trecho de código a seguir para criar a lista inicial com alguns elementos:

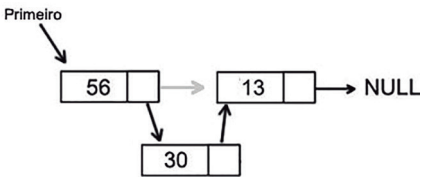
```
int main() {
Lista* listaFinal;
listaFinal = inicializar();
listaFinal = inserir(listaFinal, 13);
listaFinal = inserir(listaFinal, 56);
}
```

Assimile

É muito importante você se recordar do uso de funções, de passagem de parâmetros e ponteiros. Em uma lista ligada, o uso desses três elementos é primordial para sua criação e manuseio dentro de um programa, pois será necessário o uso de variáveis e de chamada de funções.

Na Figura 1.9, podemos observar que é possível, também, inserir um valor no meio da lista:

Figura 1.9 | Inserindo um novo elemento no meio da lista



Fonte: elaborada pelo autor.

Para isso, podemos utilizar o seguinte código:

```
Lista* inserirPosicao(Lista* l, int pos, int v){
int cont = 1;
Lista *p = l;
Lista* novo = (Lista*)malloc(sizeof(Lista));
while (cont != pos){
        p = p -> prox;
        cont++;
}
novo -> info = v;
novo -> prox = p -> prox;
p -> prox = novo;
return l;
}
```

Sua implementação de chamada da função principal pode ser definida como:

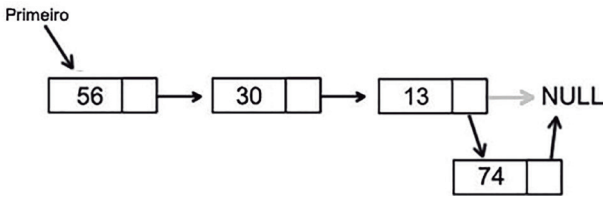
```
listaFinal = inserirPosicao(listaFinal, 1, 30);
```

Assim, no trecho implementado anteriormente, a função *inserirPosicao( )* recebe como parâmetros:

- a lista ligada (listaFinal);
- a posição que desejamos inserir, neste caso a primeira posição;
- o elemento, neste caso o valor 30.

Em uma lista, há a possibilidade de inserir um elemento no final, como mostra a Figura 1.10:

Figura 1.10 | Inserindo um novo elemento no fim da lista



Fonte: elaborada pelo autor.

Podemos visualizar a implementação da função a seguir:

```
Lista* inserirFim(Lista* l, int v){
Lista *p = l;
Lista* novo = (Lista*)malloc(sizeof(Lista));
while (p -> prox != NULL){
        p = p -> prox;
        cont++;
}
novo -> info = v;
novo -> prox = p -> prox;
p -> prox = novo;
return l;
}
```

Assim, usamos a implementação do trecho na função principal para chamar a função:

```
listaFinal = inserirFim(listaFinal, 74);
```

A chamada da função *inserirFim( )* nos permite passar como parâmetros a lista ligada (listaFinal) e o valor do elemento para inserir, no final dessa lista, o valor 74 do exemplo.

Reflita

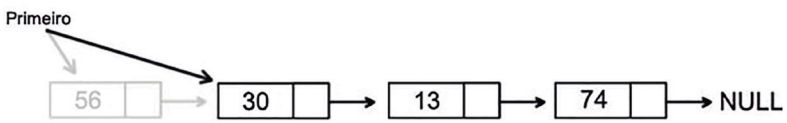
Os trechos de códigos utilizados nesses exemplos, como implementação das funções de uma lista ligada, servem como base para criar um simples programa em C++. É possível a criação de uma lista ligada sem o uso de memórias dinâmicas?

Remover elementos da lista

Com uma lista ligada criada, podemos implementar o uso de funções para a remoção de elementos da lista. Segundo Celes (2004), a função para remover um elemento é mais trabalhosa e complexa e precisa de informações como parâmetros de remoção, o valor do elemento e a lista. Assim, deve atualizar o valor da lista sem o elemento removido.

Caso o primeiro elemento da lista seja o elemento a ser retirado, devemos atualizar o valor da lista com o ponteiro para o segundo elemento, liberando o espaço alocado do elemento retirado, como podemos observar na Figura 1.11.

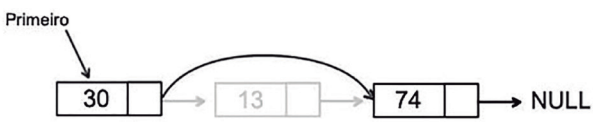
Figura 1.11 | Removendo o primeiro elemento da lista



Fonte: adaptada de Celes et al. (2004).

Se o elemento a ser retirado da lista pela função estiver no meio desta, o elemento anterior deverá apontar para o elemento seguinte do qual será removido e, depois disso, liberaremos a alocação do elemento removido, como mostra a Figura 1.12:

Figura 1.12 | Removendo o elemento do meio da lista



Fonte: adaptada de Celes et al. (2004).

Podemos utilizar um único trecho de código em uma função para realizar as operações de remoção da lista, sendo do início ou de outra posição desta.

Criamos a função e já declaramos um ponteiro inicializado em *NULL* para ser o ponteiro que receberá o elemento anterior ao que será excluído e, logo após, criamos outro ponteiro para receber a lista passada por parâmetro:

```
Lista* remove (Lista* l, int v) {

Lista* anterior = NULL;

Lista* p = l;
```

Neste momento, implementamos o comando de repetição *WHILE* para procurar o elemento a ser excluído e guardar a posição do anterior no ponteiro criado anteriormente:

```
while (p != NULL && p -> info != v) {
        anterior = p;
        p = p -> prox;
}
```

A função retorna a própria lista para a função principal, caso o elemento a ser excluído não seja encontrado na lista ligada:

```
if (p == NULL )
return l;
```

Ao ser encontrado o elemento na lista, o comando de condição *IF* verifica se é o primeiro elemento ou não da lista. Se o valor armazenado no ponteiro for NULO, então significará que é o primeiro elemento da lista; caso tenha outro valor, o elemento estará em outra posição da lista. Essa função também remove o último valor da lista, já que armazena o ponteiro do elemento anterior:

```
if (anterior == NULL) {
        l = p -> prox;
} else {
        anterior -> prox = p -> prox;
}
```

Utilizamos o RETURN da lista para retornar à função principal:

```
        return l;
}
```

Na função principal, declaramos o seguinte trecho de código para chamar a função de retirada de um elemento da lista, onde passamos por parâmetro a lista o elemento que desejamos remover:

```
listaFinal = retira(listaFinal,56);
listaFinal = retira(listaFinal,13);
```

Percorrer a lista ligada

Talvez você precise saber quais elementos fazem parte da sua lista em determinado momento do seu sistema. Para tal, é necessário percorrer toda a lista ligada para verificá-los. A lista ligada pode ser impressa com todos os seus elementos e podemos utilizar o trecho de código a seguir.

Por ser uma função que percorrerá toda a lista e de impressão em tela, podemos declará-la como *VOID*, uma função que não retornará valor para a função principal:

```
void imprimir (Lista* l) {
        Lista* p;
        printf("Elementos:\n");
```

Neste trecho, uma condição de repetição *FOR* percorre a lista e imprime todos os elementos encontrados nela:

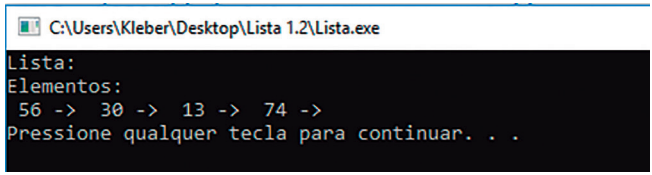
```
for (p = l; p != NULL; p = p -> prox) {
        printf(" %d -> ", p -> info);
        }
}
```

Na função principal, declaramos apenas a chamada da função *imprimir( )*, passando como parâmetro a lista na qual desejamos imprimir:

```
imprimir(listaFinal);
```

Assim, temos o resultado, que pode ser observado na Figura 1.13. Depois da lista percorrida, é impressa na tela.

Figura 1.13 | Lista impressa em tela



Fonte: elaborada pelo autor.

Verificar se um elemento se encontra na lista ligada

Outra função muito útil é verificar se determinado elemento consta na lista ou não, segundo Celes (2004). Essa função pode ser criada para receber a informação de qual elemento se deseja buscar e, caso encontre o valor, a função retorna o ponteiro do nó da lista em que representa o elemento ou sua posição na lista, ou, como no nosso exemplo, informa se o elemento foi encontrado ou não.

Podemos utilizar o trecho de código a seguir para implementar a função de busca:

```
Lista* buscar(Lista* l, int v){
    Lista* p;
```

Criamos uma condição de repetição para procurar na lista o elemento solicitado, passado por parâmetro, e, até terminar a lista, comparando se o elemento é o que está consultado. Se for verdadeiro, retornará o ponteiro; caso contrário, *NULL*:

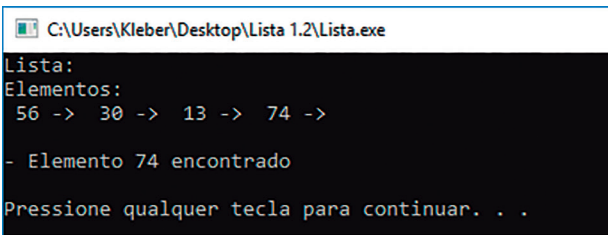
```
    for (p = l; p != NULL; p = p -> prox) {
        if (p -> info == v)
            return p;
    }
return NULL;
}
```

Já para a implementação na função principal, podemos utilizar o seguinte trecho de código para chamar a função e escrever se foi

encontrado ou não o elemento. Se o retorno for *NULL*, será escrito *Elemento não encontrado*; caso encontre, será escrito *Elemento encontrado*, como é exibido na Figura 1.14:

```
if (busca(listaFinal, 74) == NULL) {
        printf("\n Elemento não encontrado");
} else {
        printf("\n Elemento encontrado");
}
```

Figura 1.14 | Lista impressa em tela



Fonte: elaborada pelo autor.

Pesquise mais

Para complementarmos nossos estudos sobre as operações em uma lista ligada, podemos acessar o *site* da CCM sobre Listas simplesmente encadeadas no *link* a seguir. **Listas simplesmente encadeadas**. Disponível em: <http://br.ccm.net/faq/10263-listas-simplesmente-encadeadas>. Acesso em: 24 out. 2017.

Sem medo de errar

Ao retornar à situação-problema da unidade, você tem a função de atender à demanda de um cliente de autopeças que possui duas unidades, a matriz e uma filial, e deseja gerar um relatório para obter informações de estoque mínimo em seu sistema. Ele solicitou que o relatório seja gerado em tempo real, apresentando um gráfico atualizado todas as vezes que um produto entrar nessa listagem ou sair dela, por meio da venda ou compra de produtos. Esse cliente também deseja pesquisar se um produto consta na listagem ou não.

Como desafio, você precisa implementar no relatório em que está trabalhando a adição ou remoção de produtos na listagem de produtos. Como podemos adicionar ou remover produtos nessa lista? Pesquise sobre algoritmos de adição e remoção de dados em uma lista ligada.

Como é possível adicionar a função de busca por algum produto específico na listagem? Você precisará pesquisar e compreender o funcionamento do algoritmo para percorrer uma lista e verificar se determinado valor se encontra nela.

Para a resolução dessa situação-problema, é necessário compreender os conceitos sobre as operações em listas ligadas e pesquisar outras técnicas para criar as funções, para executar a adição e a remoção de elementos, assim como percorrer a lista e realizar uma contagem dos elementos e a criação da busca de elementos em uma lista ligada.

Para iniciar, implementamos a lista como:

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

struct listaProd {
    int codigo;
    char produto[30];
    struct listaProd* prox;
};
typedef struct listaProd Produtos;

Produtos* inicializar (void) {
    return NULL;
}

Produtos* inserir (Produtos * l, int i, char* nprod) {
    Produtos* novo = (Produtos*) malloc(sizeof(Produtos));
    novo -> codigo = i;
```

```
    novo -> produto = nprod;
    novo -> prox = l;
    return novo;
}

Produtos* retira (Produtos* l, int v) {
    Produtos* ant = NULL;
    Produtos* p = l;
    while (p != NULL && p -> codigo != v) {
        ant = p;
        p = p -> prox;
    }
    if (p == NULL )
        return l;
    if (ant == NULL) {
        l = p -> prox;
    } else {
        ant -> prox = p -> prox;
    }
    return l;
}

Produtos* busca(Produtos* l, int v){
    Produtos* p;
    for (p = l; p != NULL; p = p -> prox) {
        if (p -> codigo == v)
            return p;
    }
    return NULL;
}
```

Assim, podemos implementar a função principal *Main* com a seguinte solução:

```
int main() {

        int cont, codprod;

        char nprod[30];


        Produtos* listaProdutos;

        listaProdutos = inicializar(); /* inicializa lista como vazia */

        for (cont = 0; cont < 3; cont++){

                printf("\nInforme o codigo do Produto: ");

                scanf("%d",&codprod);


                printf("\nInforme o nome do Produto: \n");

                scanf("%d",&nprod);

                listaProdutos = inserir(listaProdutos, codprod, nprod);

        }


        printf("Lista Produtos:\n");

        imprimir(listaProdutos);


        printf("\nInforme o codigo do produto para pesquisa: ");

        scanf("%d", &codpro);

        if (busca(listaProdutos, codprod) == NULL) {

                printf("\n\n- Produto não encontrado\n");

        } else {

                printf("\n\n- Produto encontrado\n");

        }

        printf("\n"); system("PAUSE");

}
```

Após criar esse algoritmo, com o material desta seção, você poderá desenvolver seu algoritmo, aprimorando as funções apresentadas em aula.

Avançando na prática

Livros para leitura

Descrição da situação-problema

Diversas pessoas com o hábito de leitura têm o propósito de ler uma quantidade de livros por ano, adquirindo mais conhecimentos. Esses livros podem ser sobre qualquer assunto, recomendados por outras pessoas ou não.

Com esse pensamento, um amigo lhe pediu que o ajudasse a criar uma solução para não ficar mais anotando esses livros em um caderno. Sempre que precisasse saber se um livro estava na sua lista de leitura, tinha de procurá-lo no caderno, linha por linha, ou quando terminava de ler um livro, tinha de riscar o caderno de anotações para remover os já lidos.

Pois bem, sua tarefa é auxiliar seu amigo nessa questão.

Resolução da situação-problema

Para auxiliar seu amigo nessa tarefa, será necessário criar um sistema para que ele possa informar manualmente quais informações deseja inserir ou remover da listagem, assim como qual livro deseja buscar.

Na implementação desse programa, deverão ser adicionadas telas para seu amigo informar o livro que deseja adicionar, se será o próximo (primeiro elemento) ou o último ou que posição terá em suas prioridades. Assim, você precisará declarar a lista ligada e implementar as seguintes funções:

- inserir um livro na lista, no início, meio ou fim;
- remover um livro da lista;
- buscar um livro na lista;
- imprimir a lista de livros.

Para remover um livro da lista, seu amigo precisa informar qual o livro lido e o sistema deve excluí-lo e informar qual o próximo livro da lista a ser lido.

Com a implementação criada, é hora de desenvolver a função principal *Main*, por meio da qual seu amigo selecionará qual opção deseja e informará qual livro pretende incluir, remover ou pesquisar. O sistema realiza a chamada da função para cada opção escolhida.

Também é possível pesquisar determinado livro para saber se já consta nessa listagem.

Faça valer a pena

1. Podemos inserir um elemento em uma lista em três situações diferentes. Ao inserirmos uma informação na lista ligada, é imprescindível que seja atualizado o valor do ponteiro dessa lista, assim a lista ligada deverá apontar ao novo elemento da lista, segundo Celes (2004).

Dado o trecho de código a seguir:

```
Lista* inserir (Lista* l, int i) {
    Lista* novo = (Lista*) malloc(sizeof(Lista));
    novo -> info = i;
    novo -> prox = l;
    return novo;
}
```

Assinale a alternativa que define a funcionalidade do trecho de código informado:

a) Inserir um elemento no meio da lista.
b) Remover um elemento do início da lista.
c) Apagar toda a lista para recomeçar.
d) Inserir um elemento no final da lista.
e) Inserir um elemento no início da lista.

2. Segundo Celes (2004), a função para remover um elemento é mais complexa e precisa dos parâmetros de entrada, como o valor do elemento e a lista, atualizando o valor da lista sem o elemento removido. É possível remover qualquer elemento da lista ligada.

No caso de ser o primeiro elemento da lista a ser retirado, devemos atualizar o valor da lista com o ponteiro para o segundo elemento e, assim

_____.

Assinale a alternativa que apresenta a rotina correta para completar a sentença dada:

a) Retornar à posição do elemento retirado.
b) Liberar o espaço alocado do elemento retirado.
c) Liberar os elementos posteriores ao elemento excluído.
d) Retornar o valor do ponteiro do elemento excluído.
e) Liberar a lista para adicionar outro elemento.

3. A função pode ser criada para receber a informação de qual elemento desejamos e, caso encontre o valor, retorna o ponteiro do nó da lista em que representa o elemento ou sua posição na lista, ou, como no nosso exemplo, informa se o elemento foi encontrado ou não.

Com base no texto dado, assinale a alternativa em que consta o trecho de código correto referente ao texto:

a)
```
Lista* busca(Lista* l, int v){
    Lista* p;

    for (p = l; p!=NULL; p = p->prox) {
            if (p->info == v)
                    return p;
    }
  return NULL;
}
```

b)
```
void libera (Lista* l) {
    Lista* p = l;

    while (p != NULL) {
            Lista* t = p->prox;
            free(p);
            p = t;
    }
}
```

c)
```
Lista* inserir (Lista* l, int i) {
    Lista* novo = (Lista*) malloc(sizeof(Lista));
    novo -> info = i;
    novo -> prox = l;
    return novo;
}
```

```
d) void imprimir (Lista* l) {
    Lista* p;
    printf("Elementos:\n");
    for (p = l; p != NULL; p = p -> prox) {
            printf(" %d -> ", p -> info);
    }
}
e) Lista* inicializar (void) {
    return NULL;
}
```

Seção 1.3

Listas Duplamente Ligadas

Diálogo aberto

Caro aluno, nesta última seção sobre listas ligadas, você conhecerá as listas duplamente ligadas e suas funcionalidades. Também compreenderá as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas.

Poderá identificar as diferenças entre uma lista ligada simples e uma lista duplamente ligada, como funciona sua estrutura e quais as vantagens em utilizar essa estrutura dinâmica, permitindo que a utilização de listas seja mais simples de ser aplicada em sistemas mais complexos, como a criação de sistemas para controle de processos dentro do sistema operacional. Com as listas duplamente ligadas, é possível identificar qual o processo anterior e qual o próximo processo a ser executado.

Vamos retornar ao nosso desafio desta unidade. Você adquiriu mais experiência na sua função como programador e passou ao nível sênior. Seu líder o deixou como o único responsável por atender um cliente antigo da empresa de autopeças, que utiliza os sistemas no qual você trabalha de forma interligada entre a matriz e a filial.

Após a criação da nova funcionalidade, que permite gerar o relatório para exibir o estoque mínimo, seu cliente, ao perceber a facilidade gerada, lançou-lhe mais um desafio, com a certeza de que você tem todas as condições de ajudá-lo. Ele deseja que a listagem de produtos seja:

- criada de forma ordenada ao adicionar ou excluir produtos;
- seja exibida em tempo real, nas duas empresas, ao mesmo tempo e, para isso, precisará utilizar os recursos da lista duplamente ligada na resolução desse desafio.

Esse cliente precisa novamente de sua ajuda e conhecimento para a criação desse novo recurso em suas empresas.

Uma lista duplamente ligada será o assunto principal para solucionar o desafio do seu cliente.

Não pode faltar

Até o momento, você estudou as listas ligadas nas seções anteriores, que são estruturas de encadeamento de dados simples, ou seja, a ordenação sequencial de dados de forma simples entre os elementos, onde cada nó possui um ponteiro direcionando para o próximo elemento que se encontra na lista, como uma lista de prioridades.

Segundo Celes et al. (2004), com a estrutura de Listas Ligadas não há como percorrer os elementos de forma inversa na lista, indo do final até seu início. Apesar de ser possível a retirada de um elemento da lista com encadeamento simples, isso é mais trabalhoso, pois é necessário percorrer toda a lista para encontrar o elemento anterior, pois, dado o ponteiro para determinado elemento, temos acesso ao seu próximo elemento e não ao anterior.

Definição de lista duplamente ligada

Para dar continuidade a esta seção, você estudará a lista duplamente ligada, em que, segundo Celes et al. (2004), nesse tipo de estrutura de dados cada nó possui um elemento com informações, um ponteiro para seu próximo elemento e um ponteiro para seu elemento anterior.

Dessa forma, é possível acessar tanto o próximo elemento como o elemento anterior da lista, percorrendo-a também na ordem inversa, do final até o início. O primeiro elemento aponta para seu anterior *NULL* (valor nulo), assim como o ponteiro do último elemento da lista, como exibido na Figura 1.15.

Figura 1.15 | Lista duplamente ligada

Fonte: adaptada de Celes et al. (2004, p. 149).

Segundo Tenenbaum et al. (2007), um nó em uma lista duplamente ligada consiste na criação de três campos:

- um campo-elemento para a informação;
- um ponteiro direcionando para o próximo elemento;
- um ponteiro direcionando para o elemento anterior.

Na criação de uma lista duplamente ligada, é preciso criar, além do tipo de dado que será utilizado em sua implementação e o ponteiro que informa qual o próximo elemento, o ponteiro direcionando para o elemento anterior da lista, como o modelo de implementação a seguir:

```
struct lista {
        int info;
        struct lista* ant;
        struct lista* prox;
};
typedef struct lista Lista;
```

Exemplificando

Exemplo de declaração para criação de uma lista duplamente ligada em C:

```
struct Pessoa {
    char nome[50];
    char sexo;
    int idade;
    struct Pessoa* ant;
    struct Pessoa* prox;
};
typedef struct pessoas Pessoa;
```

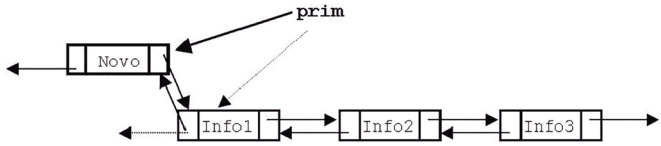
Como na lista ligada simples, também precisamos inicializar a lista duplamente ligada para a utilizarmos após a sua declaração. Uma das possíveis formas de inicialização é criar a função, retornando a lista como nula:

```
/* Função para retornar uma lista vazia */
Lista* inicializa (void)
{
        return NULL;
}
```

Adicionar elementos à lista duplamente ligada

Podemos adicionar um novo elemento à lista duplamente ligada. Se a lista estiver vazia, esse elemento terá como elementos anterior e próximo o valor *NULL*. No caso de a lista estar com elementos inseridos, ao adicionar um novo elemento, o elemento antigo passará a ser o próximo elemento da lista e o anterior passará a receber o valor *NULL*. A Figura 1.16 representa a adição de um novo elemento no início da lista.

Figura 1.16 | Adição de um novo elemento no início da lista



Fonte: adaptada de Celes e Rangel (2004, p. 150).

O trecho de código a seguir representa a inserção de um novo elemento no início da lista:

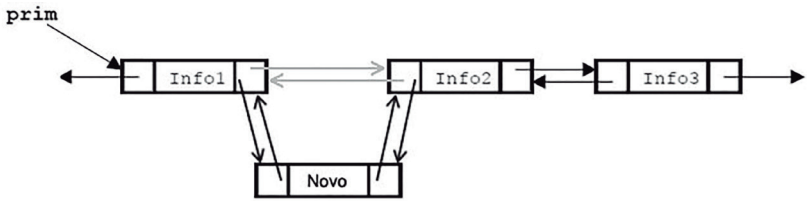
```
Lista* inserir (Lista* l, int i) {
    Lista* novo = (Lista*) malloc(sizeof(Lista));
    novo -> info = i;
    novo -> prox = l;
    novo -> ant = NULL;
    //Verifica se a lista não está vazia
    if (l != NULL)
        l -> ant = novo;
    return novo;
}
```

A chamada para a função dentro da função principal *Main* pode ser realizada pela linha:

```
int main() {
    Lista* listaFinal;
    listaFinal = inicializar();
    listaFinal = inserir(listaFinal, 20);
}
```

A adição de um novo elemento pode ser realizada tanto no início da lista como no seu final ou em uma posição específica. Na Figura 1.17, podemos notar a representação da adição de um elemento no meio da lista.

Figura 1.17 | Inserindo no meio da lista



Fonte: elaborada pelo autor.

A seguir, é apresentado um trecho do código referente à função de adição em uma posição da lista, para ser implementada por:

```
Lista* inserirPosicao(Lista* l, int pos, int v){
        int i, cont = 1;
        Lista *p = l;
        Lista* novo = (Lista*)malloc(sizeof(Lista));
        Lista* temp = (Lista*)malloc(sizeof(Lista));

        while (cont != pos){
                p = p -> prox;
                cont++;
        }

        novo -> info = v;
        temp = p -> prox;
        p -> prox = novo;
        novo -> ant = p;
        novo -> prox = temp;
        temp -> ant = novo;
        return l;
}
```

Para adicionarmos no final da lista, podemos utilizar a implementação a seguir:

```
Lista* inserirFim(Lista* l, int v){
        int cont = 1;
        Lista *p = l;
        Lista* novo = (Lista*)malloc(sizeof(Lista));

        while (p -> prox != NULL) {
                p = p -> prox;
                cont++;
        }

        novo -> info = v;
        novo -> prox = NULL;
        novo -> ant = p;
        p -> prox = novo;
        return l;
}
```

Assimile

É bom salientar que em uma lista duplamente ligada é necessário sabermos o ponteiro para o elemento anterior e para o próximo elemento, quando for implementado um código de adição de elementos no meio da lista.

Remover elementos da lista duplamente ligada

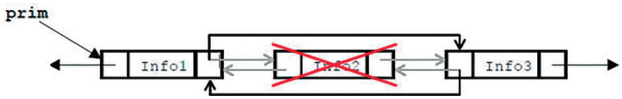
Segundo Celes et al. (2004), a função de remoção em uma lista permite remover um elemento da lista duplamente ligada apenas conhecendo o ponteiro para esse elemento. Para facilitar a localização de um elemento na lista, podemos utilizar a função de busca e, em seguida, ajustar o encadeamento da lista, por fim, liberando o elemento da alocação de memória.

Um trecho de uma função de busca pode ser implementado por:

```
Lista* busca(Lista* l, int v){
    Lista* p;
    for (p = l; p != NULL; p = p -> prox) {
        if (p -> info == v)
            return p;
    }
    return NULL;
}
```

Assim, encontrado o elemento que se deseja remover, basta apontar o anterior para o próximo e o próximo para o anterior, permitindo que o elemento no meio da lista possa ser removido do encadeamento, como na Figura 1.18:

Figura 1.18 | Remoção de um elemento da lista



Fonte: elaborada pelo autor.

Conforme Celes et al. (2004), se o elemento que desejamos remover estiver no início ou no final da lista, o apontamento para o anterior ao início será nulo, assim como se for o último, o apontamento para o próximo também será nulo.

Podemos implementar o trecho de código de remoção como segue:

```
Lista* retira (Lista* l, int v) {
    Lista* ant = NULL;
    Lista* p = l;
    while (p != NULL && p -> info != v) {
        ant = p;
        p = p -> prox;
    }
    if (p == NULL )
        return l;
```

```
        if (ant == NULL) {
                l = p -> prox;
        }
                else
        {
                p -> ant -> prox = p -> prox;
        }

        if (p -> prox != NULL)
                p -> prox -> ant = p -> ant;
        return l;
}
```

Pesquise mais

As listas duplamente ligadas também permitem a remoção de elementos do início, do meio e do final da lista, assim como a lista simplesmente ligada. No vídeo indicado a seguir é possível visualizar a implementação dessas três formas de remoção.

BACKES, A. **Aula D1 20** - Remoção na lista duplamente encadeada. Linguagem C – Programação descomplicada. 2013. Disponível em: <https://www.youtube.com/watch?v=30097hte7ys>. Acesso em: 7 nov. 2017. (vídeo do YouTube)

Ordenar a lista duplamente ligada

Em uma lista duplamente ligada, é possível realizar a ordenação de seus elementos de forma crescente ou decrescente, criando uma simples comparação entre os elementos e realizando a sua troca, em caso de aceitarem a condição de maior ou menor. Podemos utilizar o trecho de código a seguir para realizar a ordenação:

```
void Ordena(Lista* l){
  Lista* p;
  Lista* aux;
  int temp;
```

```
    for (p = l; p != NULL; p = p -> prox) {
      for (aux = p -> prox; aux != NULL; aux = aux -> prox) {
        if ((p -> info) > (aux -> info)) {
          temp = p -> info;
          p -> info = aux -> info;
          aux -> info = temp;
        }
      }
    }
  }
```

Para chamar a função de ordenação, passamos somente na função principal *Main* o parâmetro a seguir:

```
printf("\n Lista Ordenada");
Ordena(listaFinal);
```

Exemplificando

Para a utilização das funções apresentadas nesta seção, podemos criar uma função principal *Main* com a seguinte implementação:

```
int main() {
    Lista* listaFinal;
    listaFinal = inicializar();
    listaFinal = inserir(listaFinal, 68);
    listaFinal = inserir(listaFinal, 89);
    listaFinal = inserir(listaFinal, 41);
    listaFinal = inserirFim(listaFinal, 14);

    printf("Lista:\n");
    imprimir(listaFinal);

    listaFinal = retira(listaFinal, 68);
    listaFinal = inserirPosicao(listaFinal, 1, 79);
```

```
    imprimir(listaFinal); /* imprime a lista */
    printf("\n ");
    printf("\n Lista Ordenada");
    Ordena(listaFinal);
    imprimir(listaFinal);
    printf("\n");
    system("PAUSE");
    return 0;
}
```

Nesse exemplo, podemos observar que o sistema realiza a chamada da função e executa as rotinas, mas é possível, dentro da função principal *Main*, criar rotinas nas quais o usuário pode informar os elementos a serem inseridos, removidos ou realizar a busca de um elemento.

Reflita

Sabemos que as listas duplamente ligadas permitem as mesmas funções que as listas ligadas simples, no entanto podemos saber qual o elemento anterior e o próximo elemento devido aos ponteiros existentes. De que forma poderíamos imprimir a lista em ordem inversa em uma lista duplamente ligada?

Agora que você conheceu as operações sobre listas duplamente ligadas, poderá desenvolver sistemas mais complexos. Vamos praticar?

Sem medo de errar

Ao retomarmos a nossa situação-problema da seção, você adquiriu mais experiência na sua função como programador e passou ao nível sênior. Seu líder o deixou como o único responsável no atendimento de um cliente antigo, uma empresa de autopeças que utiliza sistemas que trabalham de forma interligada entre a matriz e a filial.

Após a criação da nova funcionalidade, que permite gerar o relatório para exibir o estoque mínimo, seu cliente, ao perceber a facilidade gerada, lançou mais um desafio para você, com a certeza de que possui todas as condições de ajudá-lo.

Ele deseja que a listagem de produtos seja criada de forma ordenada, ao adicionar ou excluir produtos, sendo exibida em tempo real e nas duas empresas ao mesmo tempo. Para isso, precisará utilizar os recursos da lista duplamente ligada.

Como programador, você precisa ajudar seu cliente, criando uma solução para esse desafio, para que o sistema realize a ligação das duas listas e permita a adição e exclusão mútua de produtos. Para isso, será necessário que você pesquise e compreenda como utilizar as listas duplamente ligadas. Se esse cliente abrir mais uma filial, como será possível trabalhar com as informações das três lojas?

Para resolver esse desafio, para que seu cliente implemente a ordenação dos produtos ao adicionar ou remover um produto da lista, você poderá utilizar as listas duplamente ligadas.

Com a função de ordenação de elementos, você pode implementar esse recurso no sistema, identificando de qual loja é aquele produto presente na lista e colocando todos em ordem alfabética, para facilitar a visualização.

Assim, é possível implementar a função de adição ou remoção de produtos e, na conclusão dessa função, chamar a função de **ordenação**. Ao retornar ao sistema principal, a lista de produtos já estará ordenada e atualizada.

No caso de seu cliente adquirir mais uma filial, seu sistema já estará preparado para receber os produtos com estoque mínimo desse novo empreendimento, pois já identifica de qual unidade é o produto com estoque baixo e ordena em ordem crescente, para facilitar a visualização da listagem.

A ordenação pode ser criada com base na estrutura a seguir:

```
void Ordena(Produtos* l){
   Produtos* prod;
```

```
    Produtos* aux;
    int temp;

    for (prod = l; prod != NULL; prod = prod -> prox) {
      for (aux = prod -> prox; aux != NULL; aux = aux -> prox) {
        if ((prod -> info) > (aux -> info)) {
          temp = prod -> info;
          prod -> info = aux -> info;
          aux -> info = temp;
        }
      }
    }
}
```

Assim, todas as vezes que um novo produto é inserido ou removido da listagem, a função para ordenação é chamada e executada.

Avançando na prática

Pesquisa de mercado

Descrição da situação-problema

Uma empresa que realiza pesquisa de mercado precisa solucionar um problema de demanda que surgiu. Com a alta nos valores dos produtos nos últimos meses, diversos supermercados trabalham com valores diferentes e, para entender essa diferença de valores e a margem de um mercado para outro, essa empresa contratou você, um especialista na área de programação, para desenvolver um sistema para cadastrar esses produtos de pesquisa, valores e em qual supermercado foi realizada a pesquisa. Esse sistema também deve gerar um relatório detalhado das pesquisas realizadas.

Sua função é criar esse sistema para cadastrar esses produtos, para que possam ser inseridos após o início da pesquisa e gerados relatórios com as informações detalhadas de cada produto.

Resolução da situação-problema

A criação desse sistema para a empresa de pesquisa pode nos dar um exemplo bem bacana de como criar um sistema utilizando uma lista duplamente ligada.

Podemos identificar que será necessário criar uma estrutura com um campo para descrição do produto, outro campo para descrição do supermercado no qual será realizada a pesquisa e um campo para informar o valor pesquisado, além dos ponteiros para o produto anterior e o posterior da lista:

```
struct produto {
        char[25] produto;
        char[30] mercado;
        float valor;
        struct produto* ant;
        struct produto* prox;
};
typedef struct produto Pesquisa;
```

Com base na estrutura da lista criada, criamos a implementação da função principal *Main*, em que o usuário informará o produto, o supermercado e o valor. Ao inserir esses dados na lista, um novo ponteiro para o próximo elemento e outro para o anterior serão criados. Uma das funções a serem implementadas é a de imprimir a lista duplamente ligada, de forma que o usuário possa escolher se a deseja em ordem crescente ou decrescente.

Faça valer a pena

1. Segundo Celes (2004), com a estrutura de dados das listas ligadas, não há como percorrer os elementos de forma inversa na lista, iniciando pelo fim até o início. Apesar de ser possível, a retirada de um elemento da lista com encadeamento simples é mais trabalhosa de ser realizada, pois é necessário percorrer toda a lista para encontrar o elemento anterior, pois, dado o ponteiro para determinado elemento, temos acesso ao seu próximo elemento e não ao anterior.

Com base no texto, podemos afirmar que a alternativa correta quanto às listas duplamente ligadas é:

a) Uma lista duplamente ligada requer obrigatoriamente a utilização de duas estruturas do tipo lista.

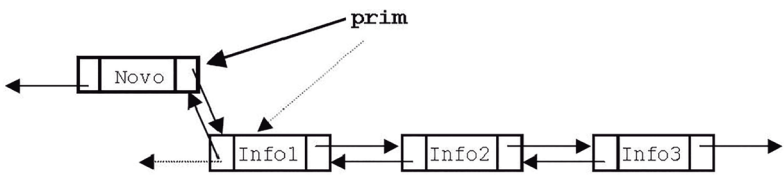
b) Uma lista duplamente ligada possui em um nó somente um ponteiro para o próximo elemento e um ponteiro para o elemento anterior.

c) Uma lista duplamente ligada não permite a impressão da lista de forma inversa, devido aos ponteiros que não mantêm a informação do elemento anterior.

d) Uma lista duplamente ligada possui em um nó o campo para informação, um ponteiro para o próximo elemento e um ponteiro para o elemento anterior.

e) Uma lista duplamente ligada, por manter informações do elemento anterior, não utiliza o recurso de alocação dinâmica de memória.

2. A Figura 1.19 representa a adição de um novo elemento no início de uma lista duplamente ligada:



Fonte: adaptada de Celes et al. (2004, p. 150).

Podemos adicionar um novo elemento à lista duplamente ligada. Se a lista estiver vazia, esse elemento terá como próximo elemento e como elemento anterior o valor *NULL*. No caso de a lista estar com elementos inseridos, ao adicionar um novo elemento, o elemento antigo passará a ser o próximo elemento da lista e o anterior passará a receber o valor *NULL*.

A alternativa que corresponde ao trecho de código que representa a adição de um novo elemento no início da lista é:

a)
```
Lista* novo = (Lista*) malloc(sizeof(Lista));
novo -> info = i;
novo -> prox = l;
novo -> ant = NULL;
```

b)
```
Lista* novo = (Lista*) malloc(sizeof(Lista));
novo -> info = i;
novo -> prox = NULL;
novo -> ant = l;
```

c)
```
Lista* novo = (Lista*) malloc(sizeof(Lista));
novo -> info = i;
novo -> prox = NULL;
novo -> ant = NULL;
```

d)
```
Lista* novo = (Lista*) malloc(sizeof(Lista));
novo -> info = NULL;
novo -> prox = l;
novo -> ant = l;
```

e)
```
Lista* novo = (Lista*) malloc(sizeof(Lista));
novo -> info = i;
novo -> prox = novo;
novo -> ant = novo;
```

3. Em uma lista duplamente ligada é possível realizar a ordenação de seus elementos de forma crescente ou decrescente, criando uma simples comparação entre os elementos e realizando sua troca em caso de aceitarem a condição de maior ou menor.

Considerando a estrutura de dados de ordenação em ordem crescente, o trecho de código que melhor define a ordenação na lista duplamente ligada é a alternativa:

a)
```
for (p = l; p != NULL; p = p -> prox) {
    for (aux = p -> prox; aux != NULL; aux = aux -> prox) {
      if ((p -> info) < (aux -> info)) {
        temp = p -> info;
        p -> info = aux -> info;
        aux -> info = temp;
}}}
```

b)
```
for (p = l; p != NULL; p = p -> prox) {
    for (aux = p -> prox; aux != NULL; aux = aux -> prox) {
      if ((p -> info) > (aux -> info)) {
        p -> info = aux -> info;
        aux -> info = p -> info;
}}}
```

c)
```
for (p = l; p != NULL; p = p -> prox) {
    for (aux = p -> prox; aux != NULL; aux = aux -> prox) {
      if ((p -> info) > (aux -> info)) {
        temp = p -> info;
        p -> info = aux -> info;
        aux -> info = temp;
}}}
```

```
d) for (p = l; p != NULL; p = p -> prox) {
     for (aux = p -> prox; aux != NULL; aux = aux -> prox) {
       if ((p -> info) < (aux -> info)) {
         p -> info = aux -> info;
         aux -> info = p -> info;
 }}}
```

```
e) for (p = l; p != NULL; p = p -> prox) {
     for (aux = p -> prox; aux != NULL; p = p -> prox) {
       if ((p -> info) > (aux -> info)) {
         temp = p -> info;
         p -> info = aux -> info;
         aux -> info = temp;
 }}}
```

Referências

CELES, W.; CERQUEIRA, C.; RANGEL, J. L. **Introdução a estrutura de dados:** com técnicas de programação em C. Rio de Janeiro: Campus Elsevier, 2004.

FORBELLONE, A. L. V. **Lógica de programação**: a construção de algoritmos e estrutura de dados. 3. ed. São Paulo: Prentice Hall, 2005.

SILVA, O. Q. **Estrutura de dados e algoritmos usando C**: fundamentos e aplicações. 1. ed. Rio de Janeiro: Ciência Moderna, 2007.

TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estrutura de dados usando C**. São Paulo: Pearson Prentice Hall, 2007.

VELOSO, P. A. S. **Estruturas de dados**. Rio de Janeiro: Campus-Elsevier, 1996.

Pilhas e filas

Convite ao estudo

Prezado aluno, daremos continuidade aos nossos estudos de Algoritmo e estrutura de dados. Aprendemos sobre listas ligadas na unidade anterior, sua definição, suas aplicações e operações, e pudemos entender o quão funcional são as listas no nosso cotidiano e em programação!

Agora, para avançarmos um pouco mais nesse conhecimento, aprenderemos sobre as definições de pilha e fila, seus elementos, seu funcionamento, as possíveis operações e aplicações em solução de problemas.

Você vai conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas. Também poderá entender as pilhas e as filas, sua construção e uso adequados, além de sua aplicação em programas de computador.

Para facilitar a compreensão desses novos conteúdos, apresentamos o seguinte contexto: a empresa de TI para a qual você trabalha o designou, por ser um dos seus melhores profissionais, para um trabalho de consultoria junto à empresa que o cliente Vinícius está abrindo. Trata-se de uma empresa que prestará um serviço terceirizado na montagem de notebooks. Todas as peças serão adquiridas prontas e, portanto, além da montagem, será necessário testar os notebooks para que saiam da empresa prontos para a venda.

Como Vinícius é um cliente importante e seu gestor conhece sua competência profissional em elaborar soluções para esse tipo de situação, ele pediu sua ajuda para criar um sistema a fim de facilitar a montagem e os testes desses notebooks.

Para realizar este desafio, você vai aprender sobre a estrutura de pilhas e filas e como funciona a adição ou a remoção de

elementos nessas estruturas, além de verificar se a pilha ou a fila está vazia ou não.

Agora que você conhece seu desafio, que tal realizar um ótimo trabalho para a empresa de Vinícius?

Então, primeiramente, vamos alicerçar nossos conhecimentos desenvolvendo os assuntos desta unidade para sermos capazes de solucionar as situações provenientes deste interessante desafio! Mãos à obra!

Seção 2.1

Definição, elementos e regras de pilhas e filas

Diálogo aberto

Caro aluno, nesta seção enfocaremos o desenvolvimento de habilidades em programação para o trabalho com pilhas e filas, que serão muito importantes na construção das competências esperadas para esta disciplina.

As pilhas são muito utilizadas para soluções de problemas, em que é necessário o agrupamento de informações, como, por exemplo, empilhar livros com a mesma letra alfabética. No caso das filas, podemos solucionar problemas em que a quantidade de informação é vasta, criando filas para aumentar o fluxo de dados.

Como vimos, seu gestor o designou para uma importante tarefa na empresa do cliente Vinícius, e você terá que ajudá-lo, criando uma solução para seu sistema de montagem e testes de notebooks. Lembre-se de que, para a realização das montagens, todas as peças serão adquiridas prontas. E, além da montagem, será necessário testar os notebooks para que saiam da empresa prontos para a venda.

Como primeiro desafio, você deve estruturar o funcionamento da montagem dos notebooks, por meio de uma fila de peças em uma esteira de produção e do empilhamento das peças em um notebook, assim como o sistema para testar os notebooks já montados pela produção criada. Para tanto, pesquise como funcionam as regras para criação da linha de produção e desenvolva este sistema com base em filas e pilhas, presentes no conteúdo estudado.

Não se esqueça de estudar os assuntos desta seção e ler atentamente as dicas nos itens pedagógicos. Faça as atividades, crie o relatório com base em seu desafio e busque se aprofundar nos conhecimentos que serão importantes para sua vida profissional!

Vamos começar?

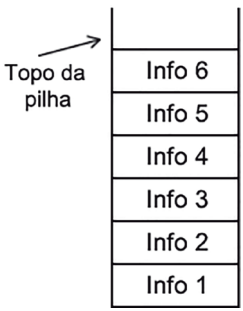
Não pode faltar

Definição e elementos de pilhas

Caro aluno, em programação é fundamental o uso de estruturas de dados para a criação de sistemas, e a mais simples e utilizada é a estrutura de dados do tipo pilha.

Segundo Tenenbaum, Langsam e Augenstein (2007), uma pilha tem como definição básica um conjunto de elementos ordenados que permite a inserção e a remoção de mais elementos em apenas uma das extremidades da estrutura denominada topo da pilha, como exibido na Figura 2.1.

Figura 2.1 | Modelo de estrutura de dados pilha.



Fonte: elaborada pelo autor.

Assim, um novo elemento que é inserido passa a ser o topo da pilha, e o único elemento que pode ser removido da pilha é o que está no topo.

Conforme Tenenbaum, Langsam e Augenstein (2007), uma pilha é um objeto dinâmico em constante mutação. Ela é mutável porque novos itens podem ser inseridos no topo da pilha, assim como o elemento que estiver nesse topo pode ser removido.

Exemplificando

Para compreendermos melhor seu funcionamento, imagine em um supermercado os empilhamentos de produtos que são colocados no início das prateleiras ou no meio dos corredores. Quando vamos comprar um produto colocado nesse empilhamento, conseguimos

apenas pegar o produto que está sempre por cima. E quando o funcionário vai abastecer, ele consegue somente inserir outros produtos no topo da pilha, como pode ser observado na Figura 2.2.

Figura 2.2 | Pilha de caixas

Fonte: <https://www.istockphoto.com/br/en/vector/boxes-on-wooded-pallet-vector-flat-warehouse-cardboard-parcel-boxes-stack-front-view-gm658846064-120259799 >. Acesso em: 29 nov. 2017.

Para acessarmos algum elemento que esteja no meio da pilha ou no final dela, precisamos remover os demais elementos colocados acima deste elemento, até encontrá-lo.

Podemos fazer uma analogia muito comum de estrutura de dados do tipo pilha com uma pilha de pratos. Quando vamos a um restaurante para comer e pegamos um prato, este está no topo da pilha. Se desejarmos pegar o prato debaixo:

- removemos o prato do topo;
- retiramos o prato debaixo;
- retornamos o prato anterior para a pilha.

A criação de uma pilha simples em C++ pode ser declarada com:

- uma estrutura de dados contendo dois elementos apenas;
- um vetor para armazenar os elementos da pilha;
- uma variável do tipo inteiro para armazenar a posição atual do topo da pilha.

A principal diferença entre uma pilha e um vetor está na forma de utilização dessas estruturas. Enquanto um vetor declaramos com um tamanho fixo, a pilha declaramos com um tamanho dinâmico que está sempre mudando, conforme os elementos são inseridos ou removidos na pilha, por meio da alocação dinâmica de memória.

Regras para operação de pilhas

A Torre de Hanói é um brinquedo pedagógico que representa uma pilha e é muito utilizado em estrutura de dados, como exemplo de programação do tipo pilha, conforme exibido na Figura 2.3.

Figura 2.3 | Torre de Hanói

Fonte: <http://www.istockphoto.com/br/foto/towers-of-hanoi-gm657122632-119714879>. Acesso em: 29 nov. 2017.

A Torre de Hanói visa transferir os elementos de uma pilha para outra, sempre mantendo o elemento de maior tamanho abaixo dos outros menores. Assim, todos os elementos devem ser movidos de uma pilha para outra, executando-se a transferência de um elemento por vez e seguindo-se essa regra.

Reflita

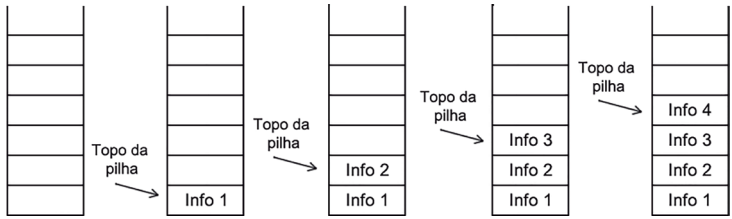
Como sabemos, a estrutura de dados do tipo pilha representa o empilhamento de elementos em uma estrutura de alocação dinâmica de memória, sendo possível identificarmos o elemento do topo. Como seria possível a contagem de elementos nessa pilha?

Os elementos inseridos em uma pilha possuem uma sequência de inserção. O primeiro elemento que entra na pilha só pode ser removido por último, após todos os outros elementos serem removidos.

Segundo Celes, Cerqueira e Rangel (2004), os elementos da pilha só podem ser retirados na ordem inversa da ordem em que nela foram inseridos. Isso é conhecido como LIFO (*last in, first out*, ou seja, o último que entra é o primeiro a sair) ou FILO (*first in, last out*, ou seja,

o primeiro que entra é o último a sair). Podemos observar a inserção na pilha do elemento sempre em seu topo, conforme a Figura 2.4:.

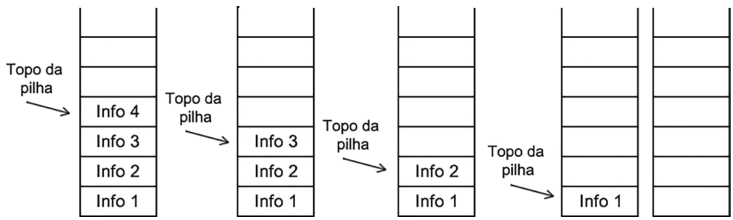
Figura 2.4 | Inserindo elemento na pilha



Fonte: elaborada pelo autor.

Também podemos remover os elementos da pilha, com identificação do elemento que está no seu topo, conforme a Figura 2.5.

Figura 2.5 | Removendo um elemento da pilha



Fonte: elaborada pelo autor.

Uma pilha também pode estar no estado de pilha vazia quando não houver elementos. Segundo Celes, Cerqueira e Rangel (2004), os passos para a criação de uma pilha são:

- criar uma pilha vazia;
- inserir um elemento no topo;
- remover o elemento do topo;
- verificar se a pilha está vazia;
- liberar a estrutura de pilha.

Definição e elementos de filas

Outra estrutura de dados muito importante é a do tipo fila, que é muito utilizada para representar situações do mundo real.

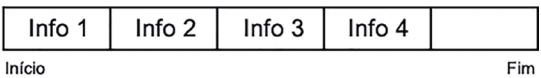
Segundo Tenenbaum, Langsam e Augenstein (2007), uma fila é a representação de um conjunto de elementos no qual podemos

remover esses elementos por uma extremidade chamada de início da fila. Já a outra extremidade, onde são inseridos os elementos, é conhecida como final da fila.

Assim como uma pilha, as filas são estruturas dinâmicas com tamanho variável, podendo aumentar ou diminuir, conforme são inseridos ou removidos elementos nelas.

Com certeza, você já enfrentou uma fila no supermercado, em uma festa para comprar fichas de alimentação ou em bancos para pagamento de uma fatura. As filas em programação são exatamente dessa forma: uma representação dessas filas do mundo real, criadas no mundo computacional, conforme podemos observar na Figura 2.6.

Figura 2.6 | Exemplo de uma estrutura de dados do tipo fila vazia



Fonte: elaborada pelo autor.

Exemplificando

Podemos citar como exemplo um aeroporto, onde diversas pessoas viajam para todos os lugares do mundo. Mas, antes de viajar, é necessário despachar as malas e realizar o *check-in* nos guichês. Para tanto, você dirige-se ao final da fila, e a primeira pessoa dessa fila está na outra ponta, ou seja, no início dela, para a realização do *check-in*. Esse processo é repetido a todo instante, e os elementos estão sempre sendo inseridos ou retirados dessa fila, como o exemplo na Figura 2.7.

Figura 2.7 | Fila em guichê de aeroporto

Fonte: <https://www.istockphoto.com/br/en/vector/airport-passengers-registration-waiting-in-line-gm890786432-246763716>. Acesso em: 29 nov. 2017.

Para conhecermos os elementos que estão em uma fila, precisamos percorrê-la, para identificar quais os elementos presentes. Assim, para criarmos uma fila, é necessário:

- um vetor dinâmico;
- uma variável do tipo inteiro para o início;
- uma variável do tipo inteiro para o fim.

Regras para operação de filas

Podemos imaginar a fila em uma agência dos Correios. Diferentemente da fila de um aeroporto para despacho de malas, a fila dos Correios é realizada com base em senhas, em que um cliente, na entrada, adquire uma senha e aguarda ser chamado por ela no painel, dirigindo-se ao guichê para atendimento.

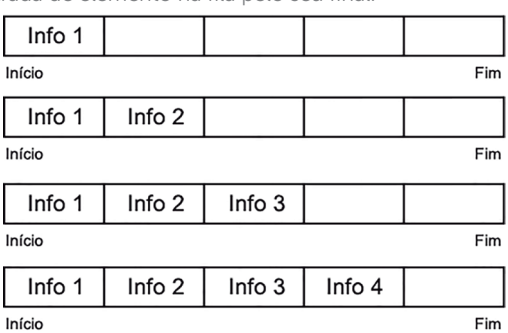
Assimile

A utilização da estrutura de dados do tipo fila difere de uma estrutura de pilha na base de sua operação, como um FIFO (*first in, first out*, ou seja, o primeiro que entra é o primeiro a sair). Assim, adicionamos novos elementos ao final da fila e removemos sempre os do seu início. Poderíamos, então, inserir um novo elemento no meio da fila?

Segundo Silva (2007), em uma fila, os elementos entram por uma extremidade e são removidos pela outra extremidade. Isso é conhecido como FIFO (*first in, first out*, ou seja, o primeiro que entra é o primeiro a sair).

No caso dessa fila, sabemos quais os elementos com base em seu número de índice, que são as senhas sequenciais. Assim, a fila possui sua ordem de entrada (fim da fila) e sua ordem de saída dos elementos (início da fila), como podemos observar na Figura 2.8.

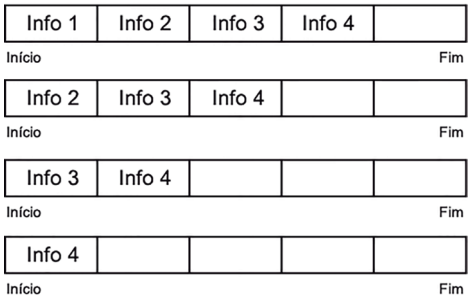
Figura 2.8 | Entrada de elemento na fila pelo seu final.



Fonte: elaborada pelo autor.

E a remoção dos elementos dá-se pela extremidade contrária, como na Figura 2.9.

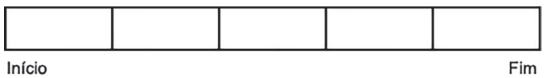
Figura 2.9 | Saída de elemento pelo início da fila



Fonte: elaborada pelo autor.

Uma fila pode estar no estado de vazia quando não contiver nenhum elemento, conforme a Figura 2.10.

Figura 2.10 | Fila vazia



Fonte: elaborada pelo autor.

Segundo Celes, Cerqueira e Rangel (2004), os passos para a criação de uma fila são:

- criar uma fila vazia;
- inserir elemento no final;
- retirar um elemento do início;
- verificar se a fila está vazia;
- liberar a fila.

Pesquise mais

Para um melhor entendimento de estrutura de dados dos tipos pilha e fila, acesse o vídeo indicado a seguir. Nele, podemos conhecer um pouco mais sobre as pilhas e filas.

UNIFACS – Universidade Salvador. Estrutura de dados – Pilhas e filas. Disponível em: <https://www.youtube.com/watch?v=RMSDm-Rgavk>. Acesso em: 21 jan. 2018. **(Vídeo do YouTube)**

Sem medo de errar

Como desafio desta unidade, seu gestor o designou para uma importante tarefa na empresa do cliente Vinícius, e você terá que ajudá-lo, criando uma solução para seu sistema de montagem e testes de notebooks.

Para a montagem, todas as peças serão adquiridas prontas. Além da montagem, será necessário testar os notebooks para que saiam da empresa prontos para a venda.

Como primeiro desafio, será necessário que você estruture o funcionamento da montagem dos notebooks e do sistema para testar os equipamentos já montados. Para tanto, pesquise como desenvolver esse sistema com base em filas e pilhas, presentes no conteúdo estudado.

Para resolver essa situação, temos como base fundamental o estudo desta seção, que vai auxiliá-lo a estruturar o funcionamento do sistema de montagem dos notebooks. Como as peças já estão prontas, precisamos apenas encaixá-las de forma que sua montagem seja concluída conforme as especificações.

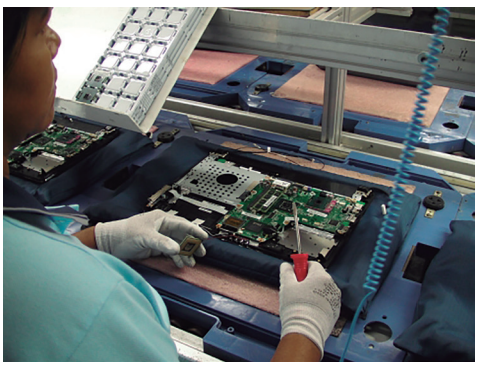
A primeira parte da estruturação do sistema pode ser iniciada pela fila de peças que serão utilizadas na montagem. Utilizando seu conhecimento adquirido em filas, você vai estruturar as esteiras com as peças, de modo que a sequência dessas peças seja disponibilizada na ordem da montagem do notebook, como o exemplo da Figura 2.11.

Figura 2.11 | Fila de montagem de notebooks

Fonte: <http://www.gazetadopovo.com.br/economia/positivo-informatica-registra-lucro-liquido-de-r-233-milhoes-em-2014-9g7igpwimvtew401m4kqf0t69>. Acesso em 29 nov. 2017.

Com as peças já sendo disponibilizadas na ordem de montagem, você pode aplicar seu conhecimento adquirido em pilhas para realizar o encaixe das peças, começando com as estruturas e ajustando essas peças uma em cima da outra, até a última peça, concluindo a montagem com a tampa final do notebook, como na Figura 2.12.

Figura 2.12 | Empilhamento de peças na montagem de um notebook



Fonte: adaptada de: <https://noamazonaseassim.com.br/no_amazonas_e_assim-13/>. Acesso em: 29 nov. 2017.

Assim, a estrutura de linha de montagem dos notebooks na ordem está realizada, disponibilizando ao seu final os notebooks prontos para testes em fila, após a montagem.

Avançando na prática

Organizando a biblioteca

Descrição da situação-problema

Imagine que a unidade da Kroton na qual você se formou está reestruturando toda a biblioteca com troca de prateleiras e nova disponibilização dos livros para os alunos. Sabendo do seu destaque como aluno e do seu grande conhecimento em estrutura de dados, a direção da faculdade solicitou seu auxílio na organização dos livros da nova biblioteca, de forma que seja mais simples e prática a disponibilização do acervo, para busca e consulta dos exemplares. Assim, você irá precisar elaborar um relatório para facilitar a organização utilizando as estruturas de pilha e fila.

Você precisará do seu conhecimento em estrutura de dados em pilha e fila, para ajudá-lo nessa organização dos livros.

Resolução da situação-problema

A reorganização dos livros para a nova disponibilização do acervo da biblioteca será realizada por você com o levantamento da estrutura de dados de pilha e fila, necessário neste desafio.

Ao remover os livros das prateleiras, será necessário organizá-los por ordem alfabética. Assim, todos os livros cujos títulos iniciam-se pela mesma letra serão colocados em uma pilha específica, como no modelo da Figura 2.13.

Figura 2.13 | Empilhamento de livros

Fonte: <http://www.istockphoto.com/br/vetor/alfabeto-de-uma-pilha-de-livros-colorido-gm477556497-35864160>. Acesso em: 29 nov. 2017.

Após todos os livros estarem separados e organizados dentro da letra em questão por ordem alfabética, você irá colocar as pilhas em uma fila para serem disponibilizadas nas prateleiras, de modo que fique cada letra em uma prateleira específica.

Faça valer a pena

1. Os elementos inseridos em uma pilha possuem uma sequência de inserção, sendo que o primeiro elemento que entra na pilha só pode ser removido por último, após todos os outros elementos serem removidos. Assim, os elementos da pilha só podem ser retirados na ordem inversa da ordem em quem foram inseridos.

O método de funcionamento de uma pilha, no qual só podem ser retirados os elementos na ordem inversa da ordem em que foram inseridos, também é conhecido como:

a) FIFO.
b) FILO.
c) FOLO.
d) FILI.
e) FOLI.

2. Uma _____ é a representação de um conjunto de elementos no qual podemos remover esses elementos por _____, chamada de início da _____, e pela outra extremidade, chamada de _____, são inseridos os elementos.

Assinale a alternativa que contém as palavras que completam a sentença anterior:

a) lista, um vetor, fila, topo da lista.
b) pilha, uma extremidade, pilha, final da pilha.
c) fila, um vetor, fila, final da lista.
d) pilha, um vetor, fila, final da fila.
e) fila, uma extremidade, fila, final da fila.

3. Segundo Tenenbaum, Langsam e Augenstein (2007), uma pilha tem como definição básica um conjunto de elementos ordenados que permite a inserção e a remoção de mais elementos em apenas uma das extremidades da estrutura denominada topo da pilha.

Com referência às pilhas, analise as sentenças a seguir:

I. É a estrutura mais simples e utilizada dentro da estrutura de dados.
II. O único elemento que pode ser removido é o que está embaixo da pilha.
III. Uma pilha é um objeto dinâmico que está em constantes mudanças.
IV. Também conhecido como FIFO (*first in, first out*).
V. A estrutura de dados pode conter dois elementos apenas, um vetor e uma variável do tipo inteiro.

Assinale a alternativa que contém as sentenças corretas:

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

Seção 2.2

Operações e problemas com pilhas

Diálogo aberto

Caro aluno, na seção anterior você estudou as definições de pilhas e filas e suas estruturas. Para darmos continuidade aos nossos estudos, nesta seção você vai conhecer as operações para inserir e remover elementos de uma pilha e como verificar se ela está vazia ou cheia.

As pilhas estão presentes no nosso dia a dia, sem que possamos perceber. Tenho certeza de que você já deve ter visto os caminhões-cegonha pelas estradas, em fotos ou até mesmo na televisão. Os carros que estão no caminhão são inseridos um de cada vez e, se for preciso remover o primeiro carro colocado no caminhão, será necessário retirar o que está atrás dele.

Com o intuito de conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas, você vai estudar a problematização sobre labirintos, desafios direcionados para soluções com pilhas.

Imagine poder utilizar a programação com a estrutura de pilhas para a solução de qualquer tipo de labirinto, com o uso de funções de inserir e remover elementos de uma pilha.

Você vai poder implementar soluções para diversos tipos de problemas utilizando pilhas, como Torre de Hanói, empilhamento de caixas em um estoque, ou até mesmo para a solução de montagem de produtos pelo método de empilhamento.

Como sabemos, você foi recomendado pela empresa de TI na qual trabalha para auxiliar o cliente Vinícius no desenvolvimento de um sistema de prestação de serviços terceirizados em montagem de notebooks. Sua função será realizar o levantamento para a linha de montagem e teste dos notebooks, com a criação de um módulo

de testes nos notebooks, utilizando as operações de pilhas. Esse sistema vai precisar testar as peças montadas e realizar sua troca, caso haja algum problema.

Você tem o desafio de realizar o levantamento do sistema de testes dos notebooks, a fim de identificar as peças com defeitos e realizar sua troca.

Vamos começar?

Não pode faltar

Na seção anterior, você estudou que uma pilha é uma estrutura de dados do tipo LIFO (*Last in, First out*), ou seja, na qual o último elemento a entrar é o primeiro a sair. Assim, nos elementos que ainda permanecem, o que está no topo da pilha será o primeiro a ser removido.

Conforme Celes, Cerqueira e Rangel (2004), em uma estrutura de Pilha, devem ser implementadas duas operações básicas:

- empilhar um novo elemento;
- desempilhar um elemento.

Segundo Lorenzi, Mattos e Carvalho (2015), a operação de empilhar um novo elemento tem a função de inserir um elemento na pilha, sendo definida na programação em C++ como *push( )*. Por exemplo, ao colocar um livro em cima de uma pilha de livros, você estará inserindo esse livro no topo da pilha.

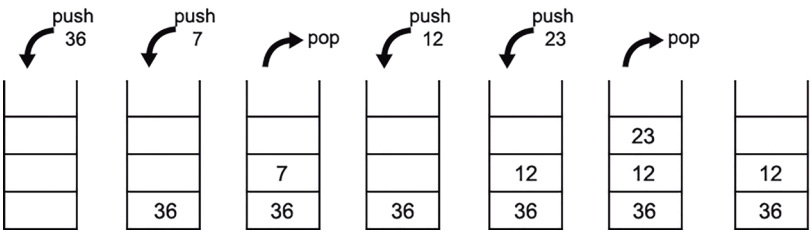
Já a operação de desempilhar tem a função de remover um elemento do topo da pilha, sendo utilizada na programação em C++ como *pop( )*. Por exemplo, remover o livro que está no topo da pilha.

Conforme Drozdek (2016), outras operações que podem ser implementadas na pilha são:

- limpar a pilha, utilizando a função *clear( )*;
- verificar se a pilha está vazia, com base na função *isEmpty( )*.

Na Figura 2.14, a seguir, podemos observar uma sequência de operações para inserir um novo elemento na pilha com a função *push( )* e remoção do elemento com a função *pop( )*.

Figura 2.14 | Sequência de funções *push( )* e *pop( )* na pilha



Fonte: adaptada de Drozdek (2016, p. 116).

Você pôde observar, na Figura 2.14, como funciona a função *push*( ), que insere um novo elemento na pilha, e a função *pop*( ), que remove o elemento do topo. No início, a pilha está vazia e, com a função *push*, é inserido o valor 36, depois, o valor 7. No próximo passo, com a função *pop*, é removido o valor 7. Novamente, com a função *push*, são inseridos os valores 12 e 23. Por fim, é removido o valor 23 da pilha.

Exemplificando

Podemos dar como exemplo a utilização de cestas de compras em um supermercado. Elas ficam empilhadas uma dentro da outra e, para utilizar uma cesta, um cliente deve retirar a cesta que está no topo dessa pilha.

Após serem recolhidas pelo supermercado por um colaborador, as cestas são inseridas no topo da pilha, onde ficam alocadas.

Inserir elementos na pilha

Conforme Tenenbaum, Langsam e Augenstein (2007), uma pilha possui uma estrutura que pode ser declarada contendo dois objetos:

- um ponteiro, que irá armazenar o endereçamento inicial da pilha;
- um valor inteiro, que irá indicar a posição do topo da pilha.

Para utilização de uma pilha, primeiramente, é necessário:

- criar a declaração da estrutura da pilha;
- criar a pilha com a alocação dinâmica;
- criar as funções para inserir na pilha e remover dela.

A declaração da estrutura inicial para criação de uma pilha pode ser implementada por:

```
struct Pilha {
        int topo;
        int capacidade;
        float * proxElem;
};
struct Pilha minhaPilha;
```

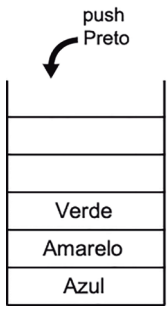
Segundo Celes, Cerqueira e Rangel (2004), com a estrutura declarada, você pode criar a função para criar uma pilha. Esta função aloca, dinamicamente, na memória o espaço para utilização da pilha.

O trecho de código para criação da pilha pode ser implementado por:

```
void cria_pilha(struct Pilha *p, int c ){
   p -> proxElem = (float*) malloc (c * sizeof(float));
   p -> topo = -1;
   p -> capacidade = c;
}
```

Com a função para criar a pilha realizada, ela estará vazia, ou seja, não terá nenhum elemento na pilha em sua criação. Assim, você pode criar a função que vai permitir ser inserido um novo elemento na pilha, como podemos observar na Figura 2.15.

Figura 2.15 | Inserindo um elemento na pilha



Fonte: elaborada pelo autor.

Observe, na Figura 2.15, que a função *pop( )* insere o elemento preto no topo da pilha, passando este elemento a ser o topo da pilha, agora.

Para inserir um novo elemento na pilha em programação, você pode criar a função com o nome *push_pilha( )*, como demonstrado um trecho para implementação a seguir:

```
void push_pilha(struct Pilha *p, float v){
        p -> topo++;
        p -> proxElem [p -> topo] = v;
}
```

No trecho apresentado, a função *push_pilha( )* recebe a *struct* da pilha e o elemento a ser inserido por meio da passagem de parâmetros pela variável *v*.

Assimile

Precisamos nos recordar de que a função para adicionar um novo elemento na pilha será realizada somente para inserir este elemento no topo da pilha, não permitindo a inserção no seu meio.

Remover elementos da pilha

Agora que você já sabe como inserir um novo elemento na pilha, vai aprender a remover um elemento do seu topo. Como já estudamos, a remoção de um elemento é realizada somente pelo topo da pilha, como o exemplo da Figura 2.16, na qual o elemento laranja está no topo e será removido com a função *pop( )*.

Figura 2.16 | Removendo elemento do topo da pilha

Fonte: elaborada pelo autor.

A implementação do trecho de código para a remoção de elementos do topo da pilha pode ser declarada como no modelo a seguir, utilizando a função *pop_pilha( )*:

```
float pop_pilha (struct Pilha *p){
   float aux = p -> proxElem [p -> topo];
   p -> topo--;
   return aux;
}
```

No trecho apresentado, a função *pop_pilha ( )* recebe como para a *struct* da pilha, e a variável *aux* declarada recebe o elemento que está no topo. Na linha a seguir, o valor do topo é decrementado, sendo retornado o elemento removido da pilha.

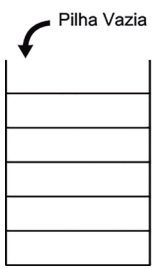
Pesquise mais

Para conhecer mais sobre as aplicações das funções com pilhas, acesse o vídeo indicado a seguir, que mostra como um jogo de cartas utiliza as técnicas de pilha. SOARES, S. Estrutura de dados - Pilha em C#. Disponível em: <https://www.youtube.com/watch?v=nHF_pgep9qE>. Acesso em: 22 jan. 2018. (**Vídeo do YouTube**)

Informar se a pilha está vazia

Uma pilha não possuirá nenhum elemento em sua inicialização, assim, não é possível remover elementos, apenas inseri-los. É interessante tratar este método para o sistema e informar ao usuário que a remoção do elemento não é possível e que a pilha está vazia, como na Figura 2.17.

Figura 2.17 | Pilha vazia



Fonte: elaborada pelo autor.

O trecho para implementação do código para verificar se a pilha está vazia pode ser dado por:

```
/*Declaração da função pilha_vazia com passagem da pilha por parâmetro*/

int pilha_vazia (struct Pilha *p ){

  if( p -> topo ==  -1 )

   return 1; /*Sendo o topo igual a -1, a função retorna verdadeiro*/

  else

    return 0; /*Caso contrário, a função retorna verdadeiro*/

}
```

O código dado verifica se a posição do topo é -1 (menos um). Caso seja, a função retorna 1 (um) como positivo para uma pilha vazia, ou 0 (zero), caso o valor do topo seja diferente, identificando, assim, que a pilha não está vazia.

Da mesma forma que podemos verificar se a pilha está vazia, é possível verificar se ela está cheia, comparando se o valor do topo é igual ao valor total da sua capacidade. O trecho a seguir apresenta a implementação do código para verificar se a pilha está cheia:

```
int pilha_cheia ( struct Pilha *p ){
      if (p -> topo == p -> capacidade - 1)
            return 1;
      else
            return 0;
}
```

Reflita

Conforme você pôde observar, em uma pilha é possível identificarmos e removermos o elemento do topo. Vamos supor que o usuário deseja saber quais os elementos da pilha e em quais posições eles estão. Como seria possível realizar essa operação?

Operações com pilhas: problema do labirinto

Um dos problemas mais comuns para solucionar com pilhas são os labirintos. Estes são desafios criados como problematização de estrutura de dados. As pilhas podem ser aplicadas também no uso de algoritmos de *Backtracking*, que consiste em criar marcações para onde o algoritmo pode retornar.

Em um labirinto, por exemplo, para encontrar um caminho correto, podemos andar pelo labirinto até encontrarmos uma divisão nesse caminho. Assim, adicionamos a posição onde a divisão ocorre, junto ao caminho escolhido na pilha, e seguimos por ele.

Caso o caminho escolhido não possua uma saída, é removido o ponto anterior da pilha, voltando ao último ponto em que o labirinto se dividiu, e recomeçamos por um outro caminho ainda não escolhido, adicionando na pilha o novo caminho.

O algoritmo de *Backtracking* pode ser aplicado também como operação de desfazer, existente em diversas aplicações de usuários, como, por exemplo, a utilização deste algoritmo em sistema de GPS. Quando o motorista utiliza uma rota não indicada pelo programa, o algoritmo de *Backtracking* é aplicado para redefinir a nova rota.

Para implementar a operação de *Backtracking*, as ações são armazenadas em uma pilha e, caso a operação de desfazer seja realizada, o estado anterior do sistema pode ser restaurado, ou a ação contrária à realizada pode ser executada.

A seguir, um trecho de implementação de criação de uma solução para labirinto:

```
/*Chama a função inicLabirinto, passando o labirinto, a pilha, o valor da linha e da coluna como passagem de parâmetros*/

void inicLabirinto(Labirinto *l, Pilha *p_l, int linha, int coluna){

  int i, j, flag = 0;

  char aux;

  elem_t_pilha origem;
```

```
/*Aplicamos uma rotina em matriz para verificar se a posição foi visitada (1) ou não (0)*/
    for(i = 0 ; i < linha ; i++){
      for(j = 0 ; j < coluna ; j++){
        if(l->p[i][j].tipo == '0'){
          l->p[i][j].visitado = 1; //Visitado
          origem.x = i;
          origem.y = j;
          /*Insere na pilha a posição de origem*/
          push(p_l, origem);
        }
      }
    }
  }
```

Assim, o algoritmo de *Backtracking* tem como meta resolver o problema no menor intervalo de tempo possível, sem levar em conta o esforço de operações para a solução do problema.

Agora é com você!

Sem medo de errar

Como sabemos, você foi recomendado pela empresa de TI na qual trabalha para auxiliar o cliente Vinícius no desenvolvimento de um sistema de prestação de serviços terceirizados em montagem de notebooks. Sua função será realizar o levantamento para a linha de montagem e o teste dos notebooks, com a criação de um módulo de testes dos notebooks utilizando as operações de pilhas. Esse sistema vai precisar testar as peças montadas e realizar sua troca, caso haja algum problema.

Com isso, você tem o desafio de realizar o levantamento do sistema de testes dos notebooks, a fim de identificar as peças com defeitos e realizar sua troca.

Você deverá entregar esse desafio em forma de relatório, com a implementação das operações de pilhas, para verificar se o notebook está montado corretamente.

Para a resolução desse desafio, será necessário montar o esquema de montagem dos notebooks e a sequência de peças que serão inseridas.

A montagem tem como base criar um sistema em que as peças serão disponibilizadas na esteira, conforme a montagem do notebook, com a primeira peça a ser colocada entrando primeiro, assim, a última peça a ser inserida no notebook será a peça do topo.

Caso algum notebook apresente alguma falha, o sistema vai identificar e desempilhar as peças colocadas até encontrar a peça com defeito para ser substituída. Ao trocar a peça defeituosa, o sistema remonta o notebook com base em sua pilha de peças, a partir da peça trocada.

```
struct peça {
        int topo; /* posição elemento topo */
        int qtd;
        float *pPeca; /* aponta para a próxima peça */
};

void criarteste(struct Pecas *p, int c ){
   p -> topo = -1;
   p -> qtd = q;
   p -> pPeca = (float*) malloc (c * sizeof(float));
}

int note_vazio ( struct Pecas *p ){
   if( p -> topo == -1 )
     return 1; /* Caso não tenha pecas, o notebook está sem pecas
*/
   else
     return 0; /* Caso contrário, o notebook está com pecas */
}
```

```
int note_completo (struct Pecas *p){
        if (p -> topo == p -> capa - 1)
                return 1; /* Caso o notebook esteja completo retorna 1 para verdadeiro */
        else
                return 0; /* Caso contrario retorna 0 para falso */
}

void insere_peca (struct Pecas *p, float num){
        p -> topo++;
        p -> pPeca [p -> topo] = num;
}

float remove_peca (struct Pecas *p){
  float aux = p -> pPeca [p -> topo]; /* Aux recebe a peca removida do topo */
   p -> topo--;
   return aux;
}

float retorna_peca (struct Pecas *p ){
  return p -> pPeca [p -> topo]; /* Retorna qual peca está no topo */
}

int main(){
        struct Pecas TestePecas;
        int qtd, op;
        float num;
        printf( "\nInforme a quantidade de pecas do notebook? " );
        scanf("%d", &qtd);
```

```
        criarteste (&TestePecas, qtd);
        while( 1 ){ /* Menu para teste */
                printf("\n1- Inserir Peca \n");
                printf("2- Remover Peca \n");
                printf("3- Mostrar ultima peca \n");
                printf("4- Sair\n");
                printf("\nopcao? ");
                scanf("%d", &op);
                switch (op){
                        case 1:
                                if(note_completo( &TestePecas )
== 1)
                                        printf("\nNotebook
Completo! \n");
                                else {
                                        printf("\nInformar o numero
da peca? \n");
                                        scanf("%f", &num);
                                        insere_peca (&TestePecas,
num);
                                }
                                break;
                        case 2:
                                if (note_vazio(&TestePecas) == 1)
                                        printf("\nNotebook Vazio!
\n");
                                else{
                                valor = remove_peca (&TestePecas);
```

```
                    printf ("\n%f Peca
removida!\n", num);
                }
                break;

            case 3:
                if (note_vazio(&TestePecas) == 1)
                    printf("\nNotebook Vazio!
\n");
                else {
                    valor = retorna_peca
(&TestePecas);
                    printf ("\nUltima peca
inserida: %f\n", num);
                }
                break;

            case 4:
                exit(0);
            default: printf("\nOpcao Invalida! \n");
        }
    }
}
```

Avançando na prática

Torre de Hanói

Descrição da situação-problema

Sua amiga Fernanda é professora de Matemática do Ensino Fundamental. Em todos os anos em que ela ministra aula, apresenta a Torre de Hanói para os alunos, a fim de que eles possam desenvolver o raciocínio lógico.

No entanto, Fernanda gostaria que seus alunos pudessem desenvolver ainda mais seu raciocínio e que conseguissem resolver a Torre de Hanói com um número maior de discos.

Assim, Fernanda solicitou a você, como profundo conhecedor de pilhas, o desenvolvimento de um algoritmo que permita a ela informar a quantidade de discos e que o sistema informe a quantidade total de movimentos que serão realizados.

Com esse algoritmo, ela poderá criar gincanas e desafios entre os alunos, para resolverem a Torre de Hanói com um número maior de discos, como na Figura 2.18.

Figura 2.18 | Esquema de uma Torre de Hanói

Fonte: <https://www.istockphoto.com/br/foto/a-torre-de-han%C3%B3i-isolada-no-branco-gm177370171-20377888>. Acesso em: 28 dez. 2017.

Podemos começar?

Resolução da situação-problema

A Torre de Hanói é um dos melhores exemplos de utilização de pilhas, com suas funções de inserir um elemento ou removê-lo da pilha.

Para o desenvolvimento desse algoritmo, será necessário o uso da recursividade, assim como o de pilhas.

Será necessário, também, um contador para realizar a contagem da quantidade de movimento, que deverá ser apresentado ao final da Torre de Hanói. Observe a seguir:

```
#include <stdio.h>

#include <stdlib.h>

int contador = 0;

void hanoi(int n, char a, char b, char c){
        if (n == 1){
             printf("Move o disco %d         } else  {
                hanoi(n - 1, a, c, b);
                printf("Move o disco %d de %c para %c\n", n, a, b);
                hanoi(n - 1, c, b, a);
            contador++;
          }
}

int main(void)
{
        int numDiscos;
        printf("Informe o numero de discos: ");
           scanf("%d", &numDiscos);
           hanoi(numDiscos, 'A', 'B', 'C');
        printf("\n\nA quantidade de movimentos foi: %d", contador);
           return 0;
}
```

Faça valer a pena

1. Segundo Lorenzi, Mattos e Carvalho (2015), a operação de empilhar um novo elemento tem a função de inserir um elemento na pilha. É definida na programação em C++ como _____. Equivale a, por exemplo, colocar um livro em cima de uma pilha de livros.

Já a operação de desempilhar tem a função de remover um elemento do topo da pilha, sendo utilizada na programação em C++ como _____. Por exemplo, equivale a remover o livro que está no topo da pilha.

Assinale a alternativa que completa as sentenças com as respectivas funções de pilha:

a) pop( ) e push( ).
b) push( ) e struct( ).
c) struct( ) e pop( ).
d) push( ) e pop( ).
e) pop( ) e struct( ).

2. Em uma estrutura de pilha, devem ser implementadas duas operações básicas: empilhar um novo elemento e desempilhar outro elemento. Conforme Tenenbaum, Langsam e Augenstein (2007), uma pilha possui uma estrutura que pode ser declarada contendo dois objetos: um ponteiro e um valor inteiro para indicar a posição do topo da pilha.

Com base nessa afirmativa, analise as sentenças a seguir:

I. Criar o vetor para armazenamento dos elementos.
II. Criar a declaração da estrutura da pilha.
III. Criar a pilha com a alocação dinâmica.
IV. Criar a função principal *Main*.
V. Criar as funções para inserir na pilha e remover dela.

Assinale a alternativa que contém as sentenças utilizadas na declaração da estrutura inicial para criação de uma pilha:

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

3. Em estrutura de dados, um dos problemas mais comuns para solucionar com pilhas são os labirintos. Estes são desafios criados como problematização de estrutura de dados. Assim, as pilhas podem ser aplicadas também no uso de algoritmos de *Backtracking*.

O uso do algoritmo de *Backtracking* consiste em:

a) criar inserção de elementos no meio da pilha.
b) criar marcações para onde o algoritmo pode retornar na pilha.
c) criar uma pilha secundária para inserir os elementos já removidos.
d) criar a estrutura para verificar se a pilha está vazia.
e) criar a estrutura para verificar se a pilha está cheia.

Seção 2.3

Operações e problemas com filas

Diálogo aberto

Caro aluno, estamos chegando ao final desta unidade sobre pilhas e filas. Na seção anterior, você aprendeu sobre as Operações e problemas com pilhas e como implementar as funções necessárias para criar um código dessa estrutura.

A estrutura de fila é amplamente usada no mundo real, sendo muito importante na organização dos elementos que a compõem. Várias empresas estão aderindo cada vez mais ao painel de senhas para a organização de suas filas de atendimento. Por meio desse sistema, as pessoas retiram uma senha e são chamadas pelo painel, fazendo com que a fila seja organizada, mas sem que elas fiquem na mesma posição da sua chegada.

Para darmos prosseguimento a nossos estudos nesta seção, você vai aprender sobre as operações de inserção, remoção e verificação se uma fila está vazia ou não.

Para a solução de desafios referentes a filas, você vai conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas.

Assim, você será capaz de solucionar problemas relacionados a filas, muito comuns em nosso dia a dia, como em supermercados, em bancos e em pedágios nas estradas. Com isso, você poderá criar soluções para serem utilizadas na redução de filas, ou até mesmo para melhor aproveitamento dessa estrutura.

Como é de seu conhecimento, você foi direcionado pela empresa de TI na qual trabalha para realizar uma consultoria ao Vinícius, a fim de ajudá-lo na criação de um sistema para uma empresa de terceirização de serviços em montagem de notebooks, onde será feita a montagem e, após a conclusão, os testes nos equipamentos.

Como foi concluída a solução para o teste do notebook e a troca de peças com defeito, agora surgiu a demanda de essas peças de reposição serem novamente enviadas para a montagem.

Os produtos que apresentam defeitos seguem para uma nova linha de produção, entrando na fila para a reposição da peça com defeito e, após a peça ser reposta, este produto deve sair da fila. Utilizando as operações sobre filas, o sistema deve informar se existem novos produtos com defeitos, ou se a fila está vazia e trabalha de forma circular.

Como você é o responsável pela consultoria e possui grande conhecimento nesse tipo de sistema, seu desafio é solucionar o problema do seu cliente e entregar um relatório ao Vinícius sobre a solução do desafio. Preparado?

Não pode faltar

Na primeira seção desta unidade, você conheceu a definição de pilhas e filas, seus elementos e suas regras de funcionamento. Segundo Celes, Cerqueira e Rangel (2004), a estrutura de fila é uma analogia ao conceito de filas que usamos no nosso cotidiano, como em um supermercado, no banco ou em um caixa eletrônico, conforme a Figura 2.19.

Figura 2.19 | Exemplo de fila em caixa eletrônico

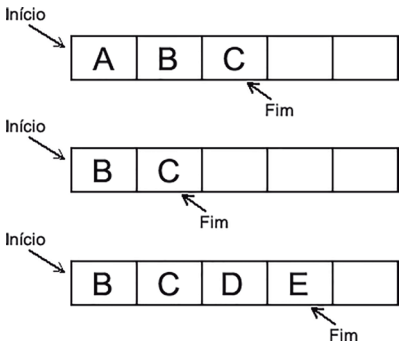
Fonte: <https://www.istockphoto.com/br/foto/grupo-de-pessoas-em-fila-para-a-caixa-gm453156437-30865510>. Acesso em: 27 dez. 2017.

Assimile

Uma lista permite a inserção e a remoção de um elemento em qualquer posição. Já a pilha permite somente a inserção e a remoção de um elemento pelo topo, enquanto a fila permite somente a remoção pelo seu início e a inserção pelo seu fim.

Você aprendeu também que a estrutura de fila é do tipo FIFO (*First in, First out*), ou seja, o primeiro elemento que entra na fila é o primeiro a ser removido. Assim, todo primeiro elemento que entra na fila por uma extremidade sairá primeiro pela outra extremidade, como o exemplo da Figura 2.20.

Figura 2.20 | Exemplo de fila em estrutura de dados



Fonte: adaptada de Tenenbaum, Langsam e Augenstein (2007, p. 208).

Nesta seção, você irá aprender a implementar uma fila como estrutura de dados. Segundo Celes, Cerqueira e Rangel (2004), a fila de impressão é um exemplo de implementação de fila muito utilizado em computação.

Pesquise mais

As filas podem ser implementadas com o uso da estrutura de vetores, também chamadas de filas estáticas. No vídeo indicado a seguir, é apresentada a implementação de criação e eliminação de uma estrutura de dados do tipo fila estática. BACKES, André. Linguagem C Programação Descomplicada. Aula 32 - Criando e destruindo uma fila estática. Disponível em: <https://www.youtube.com/watch?v=y93DzmBskGQ>. Acesso em: 22 jan. 2018. (Vídeo do YouTube)

Inserir elementos na fila

Segundo Drozdek (2016), a estrutura de dados de fila possui operações similares às da estrutura de pilha para gerenciamento de uma fila, como:

- cria uma fila vazia;
- insere um elemento no fim da fila;

- remove o elemento do início da fila;
- verifica se a fila está vazia;
- libera a fila.

Conforme Celes, Cerqueira e Rangel (2004), podemos simplesmente utilizar um vetor para armazenar os elementos e implementarmos uma fila nessa estrutura de dados ou podemos utilizar uma alocação dinâmica de memória para armazenar esses elementos.

Para iniciar uma fila, é necessário definir o tipo da estrutura a ser utilizada. Essa definição da estrutura pode ser dada pela implementação do trecho a seguir:

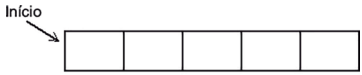
```
#define N 100

struct fila {
    int n;
    int ini;
    char vet[N];
};

typedef struct fila Fila;
```

Após a definição da estrutura, é preciso inicializar a fila, para que possa receber os elementos e, utilizando a alocação dinâmica, ser implementado o trecho de código a seguir para a inicialização da fila como vazia:

```
Fila* inicia_fila (void){
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f -> n = 0;
    f -> ini = 0;
    return f;
}
```

Podemos observar na Figura 2.21 a estrutura da fila criada e vazia.

Figura 2.21 | Estrutura de fila inicializada e vazia



Fonte: elaborada pelo autor.

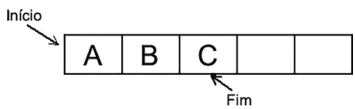
Já com a fila definida e inicializada, podemos implementar o trecho de código responsável pela inserção de elementos na estrutura criada:

```
void insere_fila (Fila* f, char elem){
        int fim;
        if (f -> n == N){
                printf("A fila está cheia.\n");
                exit(1);
        }

        fim = (f -> ini + f -> n) % N;
        f -> vet[fim] = elem;
        f -> n++;
}
```

Na função apresentada, primeiramente, passamos o ponteiro da fila e o elemento para ser inserido como parâmetros. Na estrutura de condição *IFI*, é preciso verificar se a posição do último elemento da fila é igual ao seu tamanho total. Em caso positivo, é apresentada a mensagem de que a fila está cheia e deve-se executar a saída da função.

Caso a fila não esteja completa, a função continua verificando a posição final dela, para receber o elemento a ser inserido e, assim, incrementamos nela a quantidade de elementos. Na Figura 2.22, temos a representação da fila com elementos inseridos.

Figura 2.22 | Fila com elementos inseridos



Fonte: elaborada pelo autor.

Remover elementos da fila

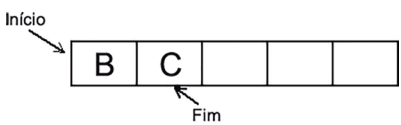
Como você já aprendeu, em uma fila só é possível remover um elemento pelo seu início. Podemos implementar o trecho de código a seguir para remoção do elemento e apresentar seu valor no retorno da função:

```
float remove_fila (Fila* f){
        char elem;
        if (fila_vazia(f)){
                printf("A Fila esta vazia\n");
                exit(1);
        }
        elem = f -> vet[f -> ini];
        f -> ini = (f -> ini + 1) % N;
        f -> n--;
        return elem;
}
```

Nesse trecho, podemos observar que, antes de removermos o elemento da fila, precisamos verificar se ela possui elementos, chamando a função *fila_vazia*. Caso a fila esteja vazia, a função apresenta uma mensagem informando ao usuário que não há elementos para serem removidos e finaliza a função.

Caso a fila possua elementos, a variável *elem* recebe o elemento da primeira posição da fila, e o início da fila passa para o próximo elemento, decrementando a quantidade de elementos e retornando o elemento removido. Na Figura 2.23, podemos visualizar que o elemento *A* foi removido e o elemento B passou a ser o primeiro da fila:

Figura 2.23 | Fila com primeiro elemento removido



Fonte: elaborada pelo autor.

Informar se a fila está vazia

Outra função que podemos implementar na fila é verificar se ela está vazia ou não. No trecho de remoção de um elemento da fila, já realizamos a chamada função, e ela pode ser implementada com o trecho de código a seguir:

```
int fila_vazia (Fila* f){
    return (f -> n == 0);
}
```

Nele, a função verifica se a quantidade de elementos da fila é igual a 0 (zero); caso seja positivo, a função retorna verdadeiro, caso contrário, retorna falso. Podemos chamar essa função sempre que precisarmos verificar se a fila está vazia ou em alguma funcionalidade em que ela não pode estar vazia para executar uma rotina específica em um sistema.

Podemos utilizar uma função importante para liberarmos a alocação de memória após o uso da fila e, assim, limparmos as variáveis utilizadas pelo sistema.

A implementação do trecho para liberação da alocação de memória pode ser dada por:

```
void libera_fila(Fila* f){
    free(f);
}
```

No trecho dado, a chamada função *free( )* libera a memória.

Reflita

É muito importante realizarmos a liberação da alocação de memória do nosso sistema. Você saberia responder quais problemas podem vir a ocorrer em um sistema operacional, se não for liberada a alocação dinâmica?

Filas circulares

Segundo Silva (2007), as filas não apresentam uma solução completa e, chegando ao final do vetor, poderemos ter a fila cheia mesmo que não esteja, uma vez que elementos podem ter sido

removidos. Para isso, podemos utilizar as filas circulares como solução para essa situação.

Uma fila circular utiliza menos instruções a serem executadas, podendo ser mais adequada e eficiente na programação. Diferentemente da estrutura de fila, a fila circular possui a seguinte definição, em sua implementação, quanto às variáveis:

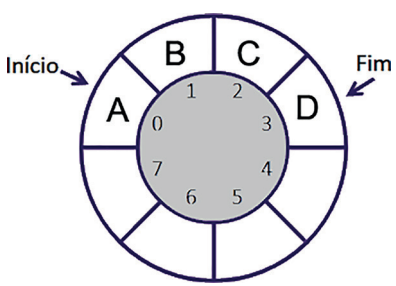
- um vetor para os elementos;
- um valor inteiro para o tamanho da fila;
- um valor inteiro para o início da fila;
- um valor inteiro para o fim da fila.

Exemplificando

Você já deve ter frequentando algum *drive-thru* (*fast food* em que o cliente não sai do carro), correto? Ao entrar na fila com seu carro, a atendente anota seu pedido e a placa do seu carro. Na próxima cabine, você realiza o pagamento, e, por fim, retira seus produtos na última cabine. Esse processo todo se encaixa no modelo de fila circular, em que os pedidos chegam a um número limite, como 999, por exemplo, e o próximo volta a ser 000.

Conforme Drozdek (2016), em uma fila circular, o conceito de circularidade se baseia no fato de o último elemento da fila estar na última posição do vetor, adjacente à primeira. Assim, são os ponteiros, e não os elementos da fila, que se movem em direção ao início do vetor, como podemos observar na Figura 2.24:

Figura 2.24 | Estrutura da fila circular



Fonte: elaborada pelo autor.

Para implementar uma estrutura de fila circular, podemos utilizar como exemplo o código a seguir:

```
/* Vamos definir a constante N com valor de 10 */
#define N 10
struct filacirc {  /* Criação da estrutura da Fila Circular */
        int tam, ini, fim;
        char vet[N];
};
typedef struct filacirc FilaCirc;

/* Função para inicializar a Fila */
void inicia_fila (FilaCirc *f){
        f -> tam = 0;
        f -> ini = 1;
        f -> fim = 0;
}

/* Função para inserir na Fila */
void insere_fila (FilaCirc* f, char elem){
        if (f -> tam == N - 1){   /* Verifica se a Fila está completa */
                printf("A fila esta cheia\n");
        } else { /* Caso a Fila não esteja completa, inserimos o elemento */
                f -> fim = (f -> fim % (N - 1)) + 1;
                f -> vet[f -> fim] = elem;
                f -> tam++;
        }
}

int fila_vazia (FilaCirc* f){
        return (f -> tam == 0); /* Retorna verdadeiro se a Fila estiver vazia */
}
```

```
char remove_fila (FilaCirc* f){
        if (fila_vazia(f)){ /* Verifica se a Fila está vazia */
                printf("Fila vazia\n");
        } else {  /* Caso a Fila contenha elemento, é removido o primeiro */
                f -> ini = (f -> ini % (N-1)) + 1;
                f -> tam--;
        }
}
```

Considerando a utilização de uma fila, alguns problemas podem surgir, por exemplo, com o uso de um vetor, haverá um armazenamento de tamanho fixo e limitado, enquanto a fila pode crescer com a necessidade de uso do sistema. Para resolver essa problemática, teríamos que limitar o tamanho máximo da fila ao tamanho do vetor.

Outra situação que pode ocorrer é adicionarmos um elemento em uma fila cheia ou removermos um elemento de uma fila vazia. Em ambos os casos, seria impossível realizar as operações. Como solução, é importante sempre implementar as funções para verificar se a fila está cheia (*fila_cheia(F)*) ou se ela está vazia (*fila_vazia(F)*).

Podem surgir problemas relacionados aos controles de início e fim de fila, em que não é possível identificar as posições nas quais se encontram. Como solução, é preciso incluir duas variáveis (*inicio* e *fim*) para armazenar a posição do início e do fim da fila, e sempre atualizar esses valores conforme a fila aumenta ou diminui.

Com essas informações, será possível criar novos algoritmos muito bem estruturados e que serão utilizados para soluções de novos problemas.

Agora é com você!

Sem medo de errar

Retomando nosso desafio, você foi direcionado para realizar uma consultoria ao Vinícius, a fim de ajudá-lo na criação do sistema para a terceirização de serviços em montagem de notebooks, em que será feita a montagem e, após a conclusão, os testes nos equipamentos.

Concluída a solução para o teste do notebook e a troca de peças com defeito, você precisa solucionar agora a demanda de as peças de reposição serem novamente enviadas para a montagem.

Os produtos que apresentam defeitos seguem para uma nova linha de produção, entrando na fila para a reposição da peça com defeito. Após a peça ser reposta, este produto deve sair da fila. Utilizando as operações sobre filas, o sistema deve informar se existem novos produtos com defeitos ou se a fila está vazia e trabalha de forma circular.

Assim, vamos pesquisar mais sobre como adicionar ou remover peças na linha de produção para atender à demanda das trocas realizadas por defeito.

Como grande conhecedor e responsável pela consultoria nesse tipo de sistema, seu desafio é solucionar o problema do seu cliente.

A fim de que o problema seja resolvido, o sistema vai precisar verificar as peças com defeitos e descartá-las. Os notebooks que apresentam alguma peça com defeito irão seguir um fluxo de filas circulares, pois, com seu retorno às esteiras de produção, precisarão ser remontados.

Com isso, os notebooks que estão na fila de produção serão montados normalmente, e aqueles cujas peças precisarem ser trocadas entrarão no final dessa fila de processo de montagem.

A remoção dos notebooks da fila de produção será realizada somente ao final de todo o processo de montagem.

É importante sempre verificar se a fila de produção não está vazia, a fim de que a esteira não fique ociosa e sem produção.

Avançando na prática

Fila de processos em um sistema operacional

Descrição da situação-problema

Devido ao seu grande conhecimento em processos, você foi contratado por uma grande empresa de desenvolvimento de sistemas para realizar a implementação de uma fila de processos em um sistema operacional. Essa fila terá um tempo para utilizar

cada um dos seus processos. No caso de um processo estar sendo executado e seu limite de tempo se encerrar, ele é removido e colocado na fila novamente; dessa maneira, o próximo processo passa a ser executado, e assim por diante, até que todos os processos tenham sido executados.

Sua função é analisar e implementar um algoritmo de uma fila circular de processos, a fim de solucionar o problema de requisições nos processos em execução, com base no tempo limite para cada processo a ser executado, e garantir o funcionamento do sistema operacional da empresa.

Seu desafio será entregar um relatório com a análise realizada e o algoritmo de como implementar essa solução junto ao seu cliente.

Resolução da situação-problema

Para a resolução desse desafio, você vai precisar pesquisar como declarar e utilizar as funções de horas (*Time*) dentro da linguagem C++.

É importante realizar um fluxograma a fim de entender como pode funcionar a fila circular para resolver o problema de processos e, com base nesse fluxograma, criar o algoritmo para a execução da solução.

A implementação do algoritmo a seguir é uma das formas possíveis de solucionar o desafio proposto, em que é verificado o tempo limite de cada processo e, assim que o tempo se esgotar, o sistema irá remover o processo da fila e passar para o próximo processo. Com base neste algoritmo:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* Declaração das funções de horas */

#define N 10
struct filacirc {
    int tam, ini, fim;
    int vet[N];
```

```
};
typedef struct filacirc FilaCirc;

void inicia_fila (FilaCirc *f){
        f -> tam = 0;
        f -> ini = 1;
        f -> fim = 0;
}

void insere_fila (FilaCirc* f, char elem){
        if (f -> tam == N - 1){
                printf("A fila esta cheia\n");
        } else    {
                f -> fim = (f -> fim % (N - 1)) + 1;
                f -> vet[f -> fim] = elem;
                f -> tam++;
        }
}

int fila_vazia (FilaCirc* f){
        return (f -> tam == 0);
}

int remove_fila (FilaCirc* f){
        if (fila_vazia(f)){
                printf("Fila vazia\n");
        } else {
                f -> ini = (f -> ini % (N-1)) + 1;
                f -> tam--;
        }
}
```

```
int main ( ){
        FilaCirc* f;
        char processo[20];
        int tempo, tmpGasto;
          clock_t tInicio, tFim; /* Declaração de variável do tipo hora
*/

          printf("\n Informe o tempo do processo em execução: \n");
            scanf("%d", &tempo);

       tInicio = clock(); /* Inicia o relógio */

        while (f -> tam < N - 1){
                insere_fila(f, processo);
        }

        while (f -> tam <= N - 1){
          tFim = clock();  /* Finaliza o relógio */
                    tmpGasto = ((int) (tFim – tInicio));  /* Calcula o
tempo gasto */

                if (tempo <= tmpGasto){  /* Se o tempo for menor
ou igual ao tempo gasto, remove da fila */
                        remove_fila(f);
                } else {
                        printf("Processando...");
        }
          system("Pause");
    }
```

Faça valer a pena

1. A estrutura de fila é do tipo FIFO (*First in, First out*), ou seja, o primeiro elemento que entra na fila é o primeiro a ser removido; assim, todo primeiro elemento que entra na fila por uma extremidade sairá primeiro pela outra extremidade.

Considerando a estrutura de dados do tipo fila, assinale a alternativa que apresenta o exemplo de uso das filas.

a) Diversos pratos um sobre o outro.
b) Anotações de tarefas a realizar.
c) Convidados de casamento.
d) Carros parados em um pedágio.
e) Torre de Hanói.

2. Em uma _____, o conceito de circularidade se baseia no fato de que o último elemento da fila está na última posição do _____, e é adjacente à primeira. Assim, são os _____, e não os elementos da fila que se movem em direção ao início do vetor.

Assinale a alternativa que apresenta as palavras que completam as lacunas.

a) pilha circular, lista, índices.
b) fila circular, vetor, ponteiros.
c) lista circular, vetor, índices.
d) lista circular, lista, ponteiros.
e) fila circular, lista, ponteiros.

3. As filas não apresentam uma solução completa. Ao final do vetor, poderemos ter a fila cheia, mesmo que ela não esteja, uma vez que elementos podem ter sido removidos. Podemos utilizar as filas circulares como solução para essa situação.

Com base na implementação das definições de fila e fila circular, assinale a alternativa que apresenta a principal diferença entre as estruturas:

a) A declaração de uma variável para o fim da fila circular.
b) A fila não permite o uso de alocação dinâmica.
c) A fila circular utiliza mais instruções na programação do que a fila.
d) A fila utiliza uma variável de controle *boolean* na estrutura.
e) A fila circular não possui uma programação adequada e eficiente.

Referências

CELES, W.; CERQUEIRA, Renato; RANGEL, José Lucas. **Introdução à estrutura de dados:** com técnicas de programação em C. Rio de Janeiro: Campus-Elsevier, 2004.

DROZDEK, A. **Estrutura de dados e algoritmos em C++**. Trad. 4. ed. norte-americana. São Paulo: Cengage Learning, 2016.

FORBELLONE, A. L. V. **Lógica de programação:** a construção de algoritmos e estrutura de dados. 3. ed. São Paulo: Prentice Hall, 2005.

LORENZI, F.; MATTOS, P. N.; CARVALHO, T. P. **Estruturas de dados**. São Paulo: Cengage Learning, 2015.

SILVA, O. Q. **Estrutura de dados e algoritmos usando C:** fundamentos e aplicações. Rio de Janeiro: Ciência Moderna, 2007.

TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estrutura de dados usando C**. São Paulo: Pearson Prentice Hall, 2007.

VELOSO, P. A. S. **Estruturas de dados**. Rio de Janeiro: Campus-Elsevier, 1996.

Tabelas de Espalhamento

Convite ao estudo

Prezado aluno, em prosseguimento aos nossos estudos em Algoritmo e Estrutura de Dados, você pôde entender e compreender na unidade anterior as definições sobre Pilhas e Listas, suas regras, funcionamento e aplicações dentro de Estrutura de Dados, bem como a solução de problemas do mundo real.

Ao avançar um pouco mais em Algoritmo e Estrutura de Dados e ampliar nosso conhecimento nesse estudo, você conhecerá e aprenderá sobre Tabela de Espalhamento, sua definição, seus elementos, funcionamento e técnicas de otimização e de colisão dentro das tabelas.

Você conhecerá e compreenderá as estruturas de dados dinâmicas essenciais e suas aplicações na solução de problemas, assim como as Tabelas de Espalhamento, sua construção e uso adequados e sua aplicação em programas de computador.

Primeiramente, você aprenderá a definição de Tabela de Espalhamento, as motivações para utilizar esse tipo de estrutura, os problemas relacionados a vocabulários e um comparativo entre as Tabelas de Espalhamento e as Listas Ligadas.

Em um segundo momento, aprenderá as operações relacionadas às Tabelas de Espalhamento, como inserir, remover e verificar a presença ou ausência de um elemento e verificar o tamanho do conjunto criado com tabelas.

Para concluir seu conhecimento em Tabelas de Espalhamento, você compreenderá a deterioração de desempenho, como aplicar as técnicas de otimização,

reduzindo e evitando as colisões e melhorando o desempenho nas Tabelas de Espalhamento.

A fim de facilitar a compreensão desses novos conteúdos a serem apresentados nesta unidade, veja a seguinte proposição: em razão do grande destaque que você e seu grupo de estudos estão tendo em programação, a unidade universitária em que estudam lhes solicitou o desenvolvimento de um novo sistema para a biblioteca da unidade.

A biblioteca está passando por uma reestruturação de melhorias e o sistema atual já não suporta mais a quantidade de livros existentes, tornando lento o retorno das informações.

Vocês terão o desafio de sanar essa dificuldade do sistema atual e desenvolver um sistema mais moderno e com melhor desempenho, a fim de que a biblioteca seja mais ágil em seu atendimento, facilitando a pesquisa de livros por categorias e por localização, em uma biblioteca maior para a disponibilização dos exemplares.

Para realizar esse desafio, você aprenderá sobre as Tabelas de Espalhamento e como funciona a adição ou remoção de elementos nessas estruturas, assim como trabalhar com as técnicas de otimização e de colisão.

Vamos começar com esse desafio e implantar um novo sistema na unidade?

Primeiramente, vamos ampliar nossos conhecimentos com base no conteúdo desta unidade e ser capazes de aplicar soluções às situações provenientes deste interessante desafio! Pronto para começar?

Seção 3.1

Definição e Usos de Tabela de Espalhamento

Diálogo aberto

Prezado aluno, na seção anterior você conheceu e aprendeu sobre as Pilhas e Filas, definições e operações como uma estrutura de dados. Nesta seção, vamos concentrar nossos estudos nas Tabelas de Espalhamento, com o intuito de desenvolvermos habilidades lógicas e de programação nessa estrutura que será de muita importância no aprendizado das competências esperadas para esta disciplina.

A estrutura da Tabela de Espalhamento é dividida em grupo de informações e, ao realizar a busca de uma informação, a estrutura elimina a maior quantidade possível de elementos que estão em outros grupos, restringindo o espaço e facilitando a busca.

Em estrutura de dados, os métodos de pesquisas existentes procuram informações armazenadas com base em comparações de chaves, como a Lista Ligada, por exemplo. Para obter um algoritmo mais eficiente, é ideal que as informações armazenadas estejam organizadas e é nesse ponto que a Tabela de Espalhamento entra.

Como vimos, você e seu grupo de amigos se destacaram em programação na unidade em que estudam e foram convidados pela faculdade a desenvolver um novo sistema para a biblioteca, substituindo o atual sistema já defasado, visto que a biblioteca está sendo reestruturada para atender à crescente demanda de alunos.

Como primeiro desafio, vocês precisaram criar um sistema moderno e com grande desempenho, atendendo à demanda de alunos e realizando buscas rápidas dos exemplares.

Será necessário utilizar a estrutura de dados com Tabelas de Espalhamento para facilitar a exposição e categorização dos exemplares.

Para isso, vocês precisarão pesquisar e realizar um levantamento de qual a melhor forma para estruturar o sistema e tornar as buscas mais rápidas para os usuários, a fim de entender e compreender a

definição e o porquê de utilizar a Tabela de Espalhamento com base no conteúdo estudado, para, ao final, entregar a melhor solução levantada em forma de projeto.

É muito importante estudar os assuntos desta seção e acompanhar as dicas presentes durante o contexto pedagógico, realizando as atividades propostas. Procure se aprofundar nos conteúdos e conhecimentos importantes para sua formação e profissão.

Preparados para começar?

Não pode faltar

Saudações, caro aluno! Você estudará nesta seção o uso das Tabelas de Espalhamento como uma estrutura de dados que permite e facilita a busca de elementos em sua estrutura.

Segundo Silva (2007), há situações, como em uma faculdade ou mesmo em um banco de código postal (CEP) dos Correios, em que a recuperação de informações em um grande volume de dados em uma tabela exige formas eficientes de acesso à informação, com buscas de melhor desempenho.

Definição de Tabelas de Espalhamento

As Tabelas de Espalhamento também são conhecidas como tabelas de dispersão, tabelas de indexação, tabelas de escrutínio, método de cálculo de endereço ou tabelas *Hash* (significa "picar", termo mais conhecido), e utilizam a técnica de endereçamento para agilizar o processo de consulta de informações.

Segundo Silva (2007), a Tabela de Espalhamento permite agilizar o processo de consulta de informações com muita eficiência, pois, para a pesquisa, não exige a ordenação sobre um conjunto de dados.

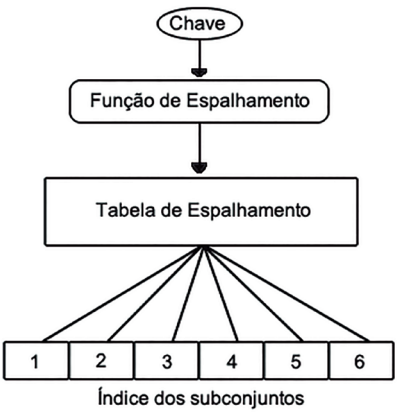
Conforme Celes et al. (2004), as estruturas de dados conhecidas como Tabelas de Espalhamento podem ser utilizadas para buscar um elemento com mais eficiência, apesar de utilizar um espaço maior de memória.

A utilização da Técnica de Espalhamento consiste basicamente em duas partes: a Função de Espalhamento e a Tabela de Espalhamento. Essa técnica permite armazenar os elementos de um

conjunto de forma espalhada pela estrutura da tabela, mas seguindo uma função lógica para o armazenamento, e não de forma aleatória ou sequencial, como nas Listas, Pilhas e Filas.

A Tabela de Espalhamento deve armazenar os subconjuntos e cada subconjunto deve ter um índice. Os índices são gerados com a Função de Espalhamento, que deve descobrir a qual subconjunto pertence determinado elemento. Para isso, deve analisar as características-chave do próprio elemento. A Função de Espalhamento também é utilizada para recuperar rapidamente o subconjunto do elemento, como pode ser observado na Figura 3.1.

Figura 3.1 | Exemplo de Tabela de Espalhamento



Fonte: elaborada pelo autor.

Exemplificando

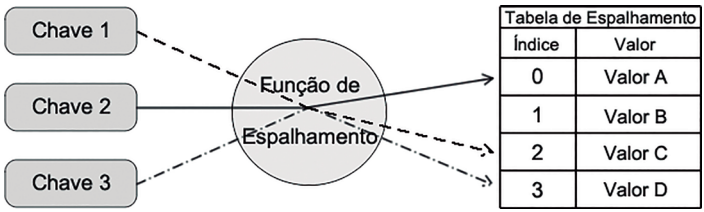
Podemos exemplificar a Função de Espalhamento no uso de uma agenda de telefone. A função deve verificar qual é a primeira letra do nome do contato solicitado e devolver o número da página referente aos contatos iniciados com a letra em questão.

Assim, temos a primeira letra como sendo a chave, usada pela Função de Espalhamento para calcular em qual índice da Tabela de Espalhamento o nome do contato será armazenado.

Para Silva (2007), a Função de Espalhamento tem como objetivo transformar o valor da chave de um elemento de dados em uma posição para esse elemento em um subconjunto criado na estrutura.

Desta forma, a Tabela de Espalhamento tem como ideia principal separar os elementos em subconjuntos, de acordo com as características-chave do próprio elemento. Desta forma, a busca de um elemento é baseada na verificação do subconjunto pertencente ao elemento para eliminar os demais subconjuntos, como se fosse realizando um filtro de pesquisas. Veja o exemplo na Figura 3.2.

Figura 3.2 | Esquema da Função e Tabela de Espalhamento



Fonte: elaborada pelo autor.

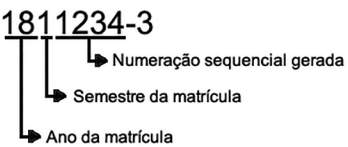
Motivação para Tabelas de Espalhamento

Por que utilizarmos as Tabelas de Espalhamento se podemos utilizar uma Lista ou um Vetor?

Para entender o uso das Tabelas de Espalhamento, vamos considerar, como exemplo, o uso de dados de alunos de uma disciplina em que cada aluno possui uma identificação numérica, como seu RA (Registro Acadêmico).

Podemos utilizar o número de RA como chave de busca para informações de cada aluno. Assim, ao buscar o número de RA no sistema, você terá acesso aos dados dos alunos cadastrados na faculdade. Vamos supor que na unidade da Kroton na qual você estuda os números de matrículas seguem o modelo 1811234-3 como padrão, sendo o dígito após o hífen um número de controle gerado pelo sistema. Neste caso, o número de matrícula efetivo seria composto dos sete primeiros dígitos: 1811234. Podemos observar na Figura 3.3 como é formado o número do RA do nosso exemplo.

Figura 3.3 | Exemplo de formação do número de registro acadêmico



Fonte: elaborada pelo autor.

Conforme Celes et al. (2004), para acessar as informações de qualquer aluno em ordem constante, apenas utilizando o número de matrícula como índice de um vetor, pode ser usado o *vet[RA]* (vetor com seu número de índex baseado no RA), de forma imediata. No entanto, para realizar essa busca, apesar de ter o acesso rápido, o consumo de memória é extremamente alto.

Neste exemplo, vamos levar em consideração que a estrutura de dados das informações associadas ao número de matrícula de cada aluno seja somente:

```
struct matricula{
    int RA;
    char nome[81];
    char email[41];
    char turma;
}
/* Define a estrutura criada como uma estrutura do tipo matrícula */
typedef struct matricula MatAluno;
```

Segundo Celes et al. (2004), para entendermos melhor o alto consumo de memória, teríamos como números de matrículas possíveis de zero a 9999999, ou seja, um vetor com dez milhões (10.000.000) de elementos.

Neste caso, a estrutura de dados para cada aluno ocuparia na memória aproximadamente 127 bytes. Como a estrutura permite dez milhões de registros, teríamos, então, 1.270.000.000 bytes, ou seja, a utilização de mais de 1 (um) gbyte de memória.

Uma das formas para diminuir o consumo de memória seria utilizar um vetor de ponteiros em vez de um vetor de estruturas. Dessa forma, quando não houver cadastro de alunos, a posição do vetor de ponteiros terá valor *NULL*. Neste caso, seriam utilizados ponteiros e a alocação dinâmica para armazenar as informações dos alunos, e acessaríamos os dados por meio de *vet[RA] -> nome*.

Para Celes et al. (2004), a utilização desta forma de uso da estrutura ocuparia 4 bytes por ponteiro, assim o gasto total seria de aproximadamente 40 mbytes, sendo considerado um consumo de memória ainda alta, apesar de ser muito menor que a forma anterior.

A forma de reduzir o consumo excessivo da memória e prover acesso rápido às informações seria com o uso das Tabelas de Espalhamento, nas quais a ideia principal é identificar na chave de busca qual a melhor forma de dividir em subgrupos as partes mais importantes.

Tendo como base a chave RA desse exemplo, podemos recordar que todos os alunos matriculados possuem um RA e todos são criados da mesma forma, com os dois primeiros dígitos do ano matriculados, o próximo e terceiro dígito informando o semestre em que eles se matricularam e os quatros últimos dígitos como números sequenciais.

Desta forma, os três primeiros dígitos não seriam utilizados para identificação individual, deixando somente os quatros últimos dígitos. O consumo de memória passa ser pequeno e o acesso aos dados do aluno permanece de forma imediata.

Assimile

Avaliar uma boa Função de Espalhamento é um trabalho que precisa ser realizado e profundamente relacionado à ideia do uso de estatística. Na maioria dos casos, a utilização de soluções mais simples, como uma Lista Ligada, pode ajudar a solucionar problemas maiores.

Problema típico de Tabelas de Espalhamento: vocabulário

De forma geral, espalhamento é o mais simples exemplo de um arranjo comum de dados, sendo uma estrutura de dados do tipo vocabulário ou dicionário. Os vocabulários são estruturas especialistas em prover as operações de inserir, remover e pesquisar.

Para Drozdek (2016, p. 473), o funcionamento desse método de busca é:

> **Diferentemente da busca sequencial, a Tabela de Espalhamento calcula a posição da chave na tabela com base no valor da chave. Esse valor é a única indicação da posição. Quando a chave é conhecida, a posição na tabela pode ser acessada diretamente, sem fazer qualquer outro teste preliminar.**

Conforme Silva (2007), a ideia geral da Técnica de Espalhamento é que, possuindo um conjunto de dados possíveis de classificação com uso de chave, podemos, então:

- dividir esse conjunto em subconjuntos, com base em critérios simples das chaves;
- identificar em qual subconjunto podemos inserir ou procurar uma chave;
- gerenciar os subconjuntos menores com métodos simples.

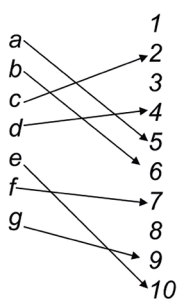
Segundo Silva (2007), a Função de Espalhamento permite identificar quantos subconjuntos serão necessários para criar uma regra de cálculo, nos quais, dada uma chave, será possível identificar em qual subconjunto será armazenada ou em qual devemos procurar as informações.

Para Celes et al. (2004), a Função de Espalhamento transforma uma chave em um endereço e associa-a ao endereço relativo do registro, apresentando as seguintes características:

- os endereços aparentam ser aleatórios, não existindo um paralelo entre a chave e o endereço, apesar de a chave ser utilizada no Espalhamento;
- é possível duas chaves direcionarem ao mesmo endereço, gerando uma colisão a ser tratada.

A ideia principal do Espalhamento é utilizar uma função com aplicação em parte da informação, chamada de chave, para retornar ao índice onde a informação deve ou deveria estar armazenada, mantendo, de forma ideal, uma informação por índice, sem repetição, como no exemplo da Figura 3.4.

Figura 3.4 | Exemplo de Espalhamento ideal



Fonte: elaborada pelo autor.

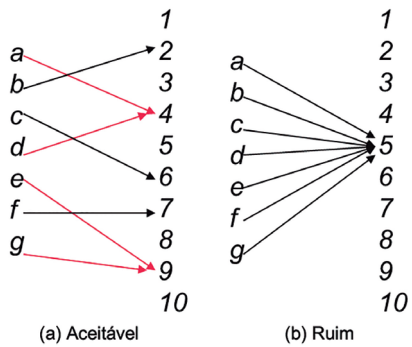
Reflita

Ao pensar em um sistema de uma empresa alimentado com informações diariamente, por diversos anos, poderíamos ter um problema com chaves, pois o número de possíveis chaves seria muito grande. Assim como teríamos um problema com espaço disponível na tabela, pois haveria muito espaço ocioso e disponível.

Poderíamos utilizar outra estrutura para armazenar essas informações? O que acarretaria tamanha quantidade de informações em apenas algumas chaves?

Na forma ideal, a função deveria gerar índices diversos para elementos diferentes, entretanto, a função pode gerar o mesmo índice para mais elementos distintos, como uma função aceitável (a), ou até mesmo como ruim (b), como podemos ver na Figura 3.5.

Figura 3.5 | Exemplo de Espalhamento não ideal



Fonte: elaborada pelo autor.

Quando ocorrem situações de um Espalhamento aceitável ou ruim, identificamos que houve uma colisão na Tabela de Espalhamento. Se voltarmos à Figura 3.5, teremos a colisão quando dois contatos com nomes diferentes se iniciam com a mesma letra.

Para resolvermos o problema de colisão, podemos, em muitos casos, utilizar outras estruturas de dados combinados com a Tabela de Espalhamento, como uma Lista Ligada. Assim, para cada elemento da Tabela de Espalhamento, teremos uma Lista ou outra estrutura de dados para armazenar mais informações na mesma posição, em vez de apenas uma informação.

Pesquise mais

Para compreender um pouco mais sobre Tabelas de Espalhamento, sua definição e seu funcionamento, você pode assistir ao vídeo indicado a seguir, que demonstra como funcionam a Tabela de Espalhamento, a Função de Espalhamento e as colisões na estrutura. TABELA de Dispersão. 2014. Disponível em: <:https://www.youtube.com/watch?v=DwDhutaOrdE>. Acesso em: 5 dez. 2017. (Vídeo do YouTube).

Comparativo: Tabelas de Espalhamento × Listas Ligadas

As Tabelas de Espalhamento, apesar de terem uma identificação grande com as Listas Ligadas, em termos de ideologia, são estruturas de dados diferentes em seus aspectos e funcionamentos, como podemos observar no Quadro 3.1.

Quadro 3.1 | Comparativo entre Tabelas de Espalhamento e Listas Ligadas

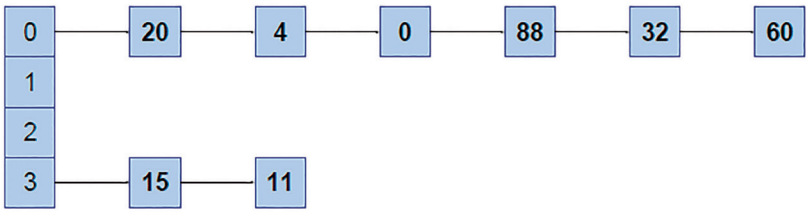
	Tabelas de Espalhamento	Listas Ligadas
Acesso aos dados	Acesso direto	Acesso sequencial
Inserção de dados	Insere com base na Função de Espalhamento por subconjuntos	Pode-se inserir dados no início, meio ou final da Lista
Remoção de dados	Remove com base na Função de Espalhamento por subconjuntos	A remoção pode ser realizada no início, meio ou final da Lista
Duplicidade de informações	Utiliza métodos para controle de colisão, podendo usar uma Lista Ligada como auxiliar para armazenamento	Não há controle de duplicidade de informação
Pesquisa de dados	Acesso direto aos dados pesquisado devido à Função de Espalhamento	Pesquisa sequencial, sendo a pesquisa realizada chave a chave

Fonte: elaborada pelo autor.

Apesar da diferença entre as estruturas, as Tabelas de Espalhamento podem utilizar Listas Ligadas, Vetores ou Árvores como estruturas auxiliares para armazenar informações.

O armazenamento e a busca de uma informação são realizados calculando-se o valor da Função de Espalhamento para a chave e direcionando esse cálculo da função para o índice da lista correspondente. Caso o tamanho das listas seja grande, a busca pode se tornar ineficiente, pois a busca nas listas se torna sequencial, como podemos observar na Figura 3.6.

Figura 3.6 | Exemplo de uma Tabela de Espalhamento com Lista Ligada



Fonte: elaborada pelo autor.

Embora permita o acesso direto ao conteúdo das informações, o uso das Tabelas de Espalhamento possui uma desvantagem em relação às Listas Ligadas, pois, apesar de realizar a indexação dos elementos, não preserva a ordem em que são disponibilizados para a Função de Espalhamento.

Agora é com você!

Sem medo de errar

Com o conhecimento adquirido nesta seção, vamos relembrar o nosso desafio. Você e seu grupo de amigos que se destacaram em programação na unidade em que estudam foram convidados pela faculdade a desenvolver um novo sistema para a biblioteca, substituindo o atual sistema, já defasado, visto que a biblioteca está sendo reestruturada para atender à crescente demanda de alunos.

Como primeiro desafio, vocês precisaram criar um sistema moderno e com grande desempenho, atendendo à demanda de alunos e realizando buscas rápidas dos exemplares.

Será necessário utilizar da estrutura de dados com Tabelas de Espalhamento para facilitar a exposição e categorização dos exemplares.

Para isso, vocês precisarão pesquisar e realizar um levantamento de qual a melhor forma de estruturar o sistema e tornar as buscas mais rápidas para os usuários. Ao final, devem entregar a melhor solução em forma de projeto.

Agora que você estudou e compreendeu o uso e o funcionamento da Tabela de Espalhamento, podemos definir um sistema de mais desempenho para uso da biblioteca.

Vamos precisar:

- estruturar a arquitetura da biblioteca para a disponibilização dos livros;
- definir como serão identificados os livros, podendo ser por base na área de conhecimento, mais o código da prateleira e, depois, o autor da obra:
  - Número decimal que corresponde ao assunto da obra na tabela de Classificação Decimal de Dewey. Ex.: 301.2.
  - Código do autor da obra, formado pela primeira letra do sobrenome do autor + número + primeira letra do título. Ex.: G562m.
- criar no sistema a Função de Espalhamento com base na chave de identificação da obra;
- criar no sistema a estrutura de dados da Tabela de Espalhamento para armazenar as obras com base na chave.

Dessa forma, poderemos ter um sistema que realizará a busca das obras com base em seu código e, assim, o acesso direto às informações da referida obra por meio do Espalhamento.

Avançando na prática

Pronto-socorro

Descrição da situação-problema

Um grande hospital da sua cidade entrou em contato com você, solicitando sua consultoria em soluções de problemas em organização de dados. Após vários estudos e grande conhecimento na área, você se tornou um exímio consultor de *software* com base em Tabelas de Espalhamento.

Esse hospital está com problemas para organizar os atendimentos do pronto-socorro. À medida que os pacientes vão chegando, as atendentes não conseguem identificar quem tem mais prioridade para atendimento e para qual médico o paciente será encaminhado.

Sua tarefa é realizar um levantamento de como esse problema pode ser solucionado para organizar e agilizar os atendimentos.

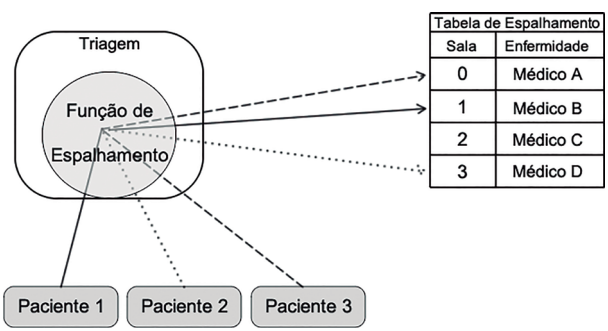
Resolução da situação-problema

Com seu grande conhecimento em solucionar esse tipo de problema utilizando as Tabelas de Espalhamento, é possível criarmos a seguinte solução:

- Implantar um sistema de senhas que serão distribuídas conforme os pacientes forem chegando.
- Implantar um sistema de triagem para identificar qual enfermidade esse paciente possivelmente possui.
- Implantar no sistema a Função de Espalhamento, a qual será atribuída uma chave com base na identificação da enfermidade e gravidade.
- Ao ser atribuída uma chave ao paciente, sua senha será inserida na lista com a chave da enfermidade.
- Cada médico atende em sua especialidade, com base na chave identificada para sua área.
- Após ser chamada pelo médico, a senha desse paciente é removida da listagem da chave identificada.

A Figura 3.7 apresenta o esquema de atendimento dos pacientes.

Figura 3.7 | Esquema de atendimento



Fonte: elaborada pelo autor.

Faça valer a pena

1. Há situações em que a recuperação de informações em um grande volume de dados em uma tabela exige formas eficientes de acesso à informação. As estruturas de dados conhecidas como Tabelas de Espalhamento podem ser utilizadas para buscar um elemento com mais eficiência, apesar de utilizar um espaço maior de memória.

Assinale a alternativa em que consta o nome mais utilizado para as Tabelas de Espalhamento:

a) Tabelas de Listas.
b) Tabelas *Hash*.
c) Tabelas de Espelhamento.
d) Tabelas-chave.
e) Tabelas de Dissipação.

2. Para resolvermos o problema de _____, podemos em muitos casos utilizar outras estruturas de dados combinados com a Tabela de _____, como uma _____. Assim, para cada elemento da Tabela de Espalhamento, teremos uma Lista ou outra estrutura de dados para armazenar mais informações na mesma _____, em vez de apenas uma informação.

Assinale a alternativa que contém as palavras que completam a sentença corretamente:

a) chave, Dispersão, Fila e posição.
b) colisão, Escrutínio, Fila e tabela.
c) chave, Espalhamento, Lista Ligada e tabela.
d) colisão, Dispersão, Pilha e posição.
e) colisão, Espalhamento, Lista Ligada e posição.

3. A ideia principal do Espalhamento é utilizar uma função com aplicação em parte da informação, chamada de chave, para retornar o índice onde a informação deve ou deveria estar armazenada, mantendo, de forma ideal, uma informação por índice, sem repetição.

Com referência às Tabelas de Espalhamento, analise as sentenças a seguir:

I. Os endereços aparentam ser aleatórios, não existindo um paralelo entre a chave e o endereço, apesar de a chave ser utilizada no espalhamento.

II. É possível duas chaves direcionarem ao mesmo endereço, gerando uma colisão a ser tratada.

III. É possível dividir este conjunto em subconjuntos com base em critérios simples das chaves.

IV. É possível identificar em qual subconjunto podemos inserir ou procurar uma chave.

V. É possível gerenciar somente os conjuntos principais com métodos simples.

Assinale a alternativa que contém as afirmativas corretas:

a) I, II, III e V apenas.

b) II, IV e V apenas.

c) I, II e III apenas.

d) I, II, III e IV apenas.

e) II, III, IV e V apenas.

Seção 3.2

Operações em Tabelas de Espalhamento

Diálogo aberto

Prezado aluno, na seção anterior você aprendeu sobre as definições e a utilização das Tabelas de Espalhamento. Para continuarmos nosso conteúdo nesta seção você aprenderá e conhecerá, as Funções de Espalhamento, operações para inserir e remover elementos de uma tabela, identificar se um elemento está presente ou ausente nela e identificar o tamanho do conjunto na Tabela de Espalhamento.

Para conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas com Tabelas de Espalhamento, você aprenderá as operações com Funções de Espalhamento e como armazenar as informações em sua estrutura.

Com o uso dessa técnica de espalhamento, será possível criar sistemas que permitam a fácil organização de informações, possibilitando, assim, a busca rápida dessa informação quando solicitada. É possível imaginarmos como seriam os sites de buscas da internet sem uma Tabela de Espalhamento para acesso direto às informações em sites de dados?

No mundo existem 13 servidores de direcionamento de sites e todos eles possuem uma tabela para armazenar todos os sites existentes e seus endereços de *IP* (*Internet Protocol* ou Protocolo de Internet), onde identificam em qual servidor do mundo encontra-se um site que você deseja acessar. Vamos supor que você irá acessar um site. Os servidores realizam a busca deste endereço do site informado em uma tabela onde constam todos os endereços de IP atribuídos a cada endereço de site no mundo, e assim ele associa o endereço que você deseja com o local onde está hospedado este site ou milhares de pessoas ao redor do mundo acessam simultaneamente esses servidores. Como seria a procura de um único endereço dentre milhares de sites nestes servidores sendo comparado linha a linha até encontrar o site desejado? Para essa solução, temos as Tabelas de Espalhamento, que permitem acessar diretamente o servidor do site que você procura, ganhando desempenho nas buscas.

Para o desafio desta seção, sabemos que seu grupo foi destaque da faculdade em programação e vocês foram convidados a desenvolver o novo sistema da biblioteca, que está sendo reestruturada em melhorias para atender à grande demanda de alunos.

O sistema atual já não suporta mais a quantidade de buscas em razão da quantidade de exemplares cadastrados, o que o deixa mais lento, além de ser um sistema antigo.

Ao definir a forma de buscas a ser realizada pelo sistema, será necessário, agora, analisar a adição e remoção de novos exemplares no sistema, além de verificar se existe ou não um exemplar no sistema da biblioteca, quando solicitado pelo aluno, com base nas Funções de Espalhamento, apresentando, após, um relatório da possível implementação da solução encontrada para esse desafio.

Pronto para solucionar esse desafio?

Não pode faltar

Caro aluno, você aprendeu que podemos utilizar as Tabelas de Espalhamento para armazenar e buscar elementos com acesso direto em sua estrutura.

Em outras estruturas, podemos utilizar operações como adicionar e remover elementos sobre as Tabelas de Espalhamento, bem como realizar a verificação da presença ou ausência de elementos e do tamanho do conjunto utilizado.

Assimile

Por meio da Função de Espalhamento, podemos obter a posição de um elemento para inseri-lo ou buscá-lo dentro da estrutura, tendo como objetivo transformar a chave em um índice na tabela.

Antes de adicionar um elemento na Tabela de Espalhamento, precisamos entender a Função de Espalhamento, que pode ser definida de duas formas principais:

- cálculo de endereços;
- divisão.

Segundo Silva (2007), ambas as funções preveem sempre o uso de uma chave simples, sendo texto (*string*) ou número no qual o cálculo será efetuado.

- Cálculo de endereços

A função de Cálculo de Endereços se baseia no armazenamento de cada entrada em um endereço calculado pela aplicação de uma função sobre o valor da chave da entrada, conforme o resultado obtido pela função sobre o valor a ordenar.

Conforme Silva (2007), podemos aplicar a Função de Espalhamento no conjunto {75, 46, 7, 34, 13}, por exemplo. A função para o Espalhamento se baseia na seguinte condição para armazenamento em uma Lista Ligada, conforme a Tabela 3.1.

Tabela 3.1 | Função de Espalhamento para cálculo de endereço

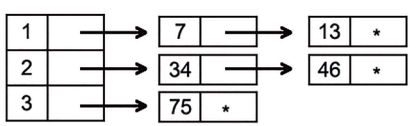
	F (valor) =
1	Se valor < 30
2	Se 30 ≤ valor < 50
3	Se valor ≥ 50

Fonte: adaptada de Silva (2007).

Conforme a Tabela 3.1, no vetor haverá um elemento para cada valor do resultado e cada um dos elementos do vetor é um índice de uma lista ligada que terá os valores dentro do intervalo definido pela função.

Segundo Silva (2007), como resultado da aplicação da função sobre o vetor dado, teremos a seguinte estrutura, conforme a Figura 3.8.

Figura 3.8 | Resultado da aplicação da Função de Espalhamento

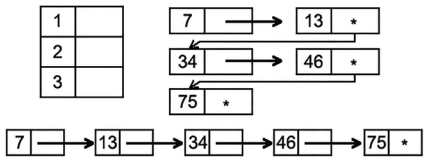


Fonte: adaptada de Silva (2007).

Segundo Silva (2007), ao serem inseridos os elementos do vetor na lista, a função verifica a estrutura para identificar em qual índice será inserido um novo valor, com base na Função de Espalhamento,

para que a lista fique ordenada. Para criar uma única lista ordenada, basta unir as listas geradas, como na Figura 3.9.

Figura 3.9 | Lista ligada ordenada com a Função de Espalhamento



Fonte: adaptada de Silva (2007).

- Divisão

Conforme Drozedek (2016), a forma de divisão é a mais simples e utilizada para a Função de Espalhamento, em que a função deve retornar um valor de índice válido para uma das células da tabela, garantindo o acesso direto aos elementos.

Para definir o endereço de um elemento na Tabela de Espalhamento, basta utilizar o resto da divisão de sua chave pela quantidade de elementos no vetor de alocação.

A divisão é dada por:

$$h(k) = mod(k,n).$$

Ou seja, a Função de Espalhamento (h) é igual ao resto da divisão (mod) entre o valor a ser buscado ou inserido (k) e a quantidade de células do vetor (n).

Podemos verificar o seguinte exemplo de aplicação dessa função:

Vamos supor que temos uma tabela de 10 posições e a seguinte sequência de chaves: 16, 65, 248, 189, 74 para ser inserida nela. Como resultado, teremos na Tabela 3.3 a seguinte distribuição na Tabela de Espalhamento, após o uso da função por divisão:

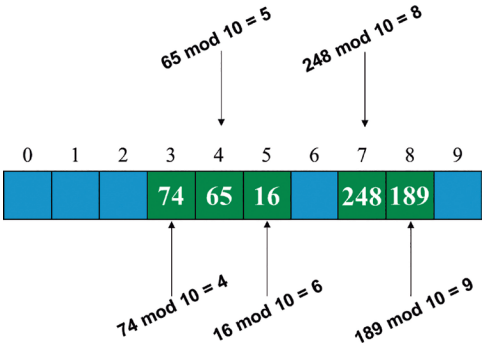
Tabela 3.3| Distribuição na Tabela de Espalhamento

Chave	Cálculo da função	Endereço
16	(16 mod 10) = 6	6
65	(65 mod 10) = 5	5
248	(248 mod 10) = 8	8
189	(189 mod 10) = 9	9
74	(74 mod 10) = 4	4

Fonte: adaptada de Silva (2007).

Representando essa tabela em forma de Tabela de Espalhamento, teremos o resultado exibido na Figura 3.10.

Figura 3.10 | Representação da Tabela de Espalhamento



Fonte: elaborada pelo autor.

Exemplificando

Para criarmos um exemplo desta seção, vamos utilizar a estrutura declarada na seção anterior:

```
#define tam 113
struct matricula{
    int RA;
    char nome[81];
    char email[41];
    char turma;
};
typedef struct matricula MatAluno;
typedef MatAluno* Hash[tam];
```

Com base nesse cálculo, podemos definir a Função de Espalhamento pelo trecho do código a seguir:

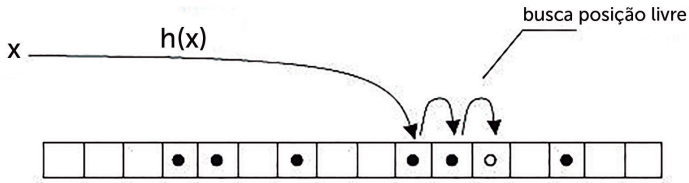
```
int funcao_Esp (int RA) {
    return (RA % tam);
}
```

Onde *tam* é a quantidade de posições na tabela.

Adição de elementos em Tabelas de Espalhamento

Conforme Celes et al. (2004), tanto na função de Cálculo de endereços como na função de Divisão, ao inserirmos um elemento na tabela e este colidir com outro elemento no endereço de índice, o elemento a ser inserido será armazenado no próximo índice disponível da própria tabela, exibido na Figura 3.11.

Figura 3.11 | Inserindo elemento na tabela com colisão



Fonte: adaptada de Celes et al. (2004).

Segundo Celes et al. (2004), uma Tabela de Espalhamento nunca será totalmente preenchida. Assim, sempre haverá uma posição disponível para armazenamento na tabela.

A função para inserir um elemento na tabela é bem simples de ser aplicada.

Para isso:

- chamamos a função para cálculo do endereço por meio da função *funcao_Esp*;
- verificamos se o elemento já está na tabela.

No caso de já existir, seu conteúdo será modificado, e, se não existir, será inserido um novo registro na tabela, com base no índice encontrado. Se o índice estiver ocupado com algum elemento, este será inserido na primeira posição livre após o índice gerado, retornando o valor do ponteiro do aluno inserido ou alterado.

Reflita

Uma Função de Espalhamento perfeita seria aquela na qual não existisse nenhuma colisão no armazenamento de informações. Sabendo-se que sempre existirão posições disponíveis na tabela, não seria possível distribuirmos os elementos, em caso de colisões, nas posições vazias? Quais seriam as consequências?

No trecho de código a seguir, podemos observar uma possível função de inserção de elementos na Tabela de Espalhamento:

```
MatAluno* insere_Esp (Hash tab, int RA, char* name, char* mail, char turma) {
    int h = funcao_Esp(RA);

    while (tab[h] != NULL) {
        if (tab[h] -> RA == RA)
            break;
        h = (h + 1) % tam;
    }

    /*Caso não encontre elemento*/
    if (tab[h] == NULL) {
        tab[h] = (MatAluno)* malloc(sizeof(MatAluno));
        tab[h] -> RA = RA;
    }

    /*Adiciona ou altera as informações na tabela*/
    strcpy(tab[h] -> nome, name);
    strcpy(tab[h] -> email, mail);
    tab[h] -> turma, turma;
    return tab[h];
}
```

Pesquise mais

Algumas funções em C++ exigem a utilização de recursos e bibliotecas adicionais da linguagem, como a função *strcpy*, que realiza a cópia de texto de uma variável para outra. Diversas são as funções existentes na biblioteca *string.h*. Para aumentar seu conhecimento nessas funções, acesse o vídeo indicado a seguir e conheça mais sobre seu funcionamento: ME SALVA! **Programação em C - PLC09 – Strings**. 2015. Disponível em: <https://www.youtube.com/watch?v=Zd-N4XDHEPY>. Acesso em: 29 jan. 2018. (Vídeo do YouTube.)

Remoção de elementos em Tabelas de Espalhamento

A remoção de um elemento da tabela pode ser realizada identificando-se qual o índice gerado pela Função de Espalhamento, para, então, ser realizada sua remoção, passando o valor *NULL* (nulo) para seu armazenamento, conforme o trecho de código a seguir:

```
void remove_Esp(Hash tab, int RA){
    int h = funcao_Esp(RA); /* A variável h recebe o valor do índice da Função de Espalhamento*/

    if(tab[h] -> RA == RA) {  /* Verifica se o RA está na tabela */
        tab[h] = NULL;   /* Caso positivo, o sistema apaga e coloca NULL no lugar */
        printf("\nRA excluido!");
    }else{
        printf("\nRA nao encontrado"); /* Caso contrário, informa que não encontrou */
    }
}
```

Verificação da presença ou ausência de elementos em Tabelas de Espalhamento

Conforme Celes et al. (2004), a operação de busca permite procurarmos a ocorrência de um elemento a partir do índice gerado pela função *funcao_Esp (int RA)*, em que o RA é a chave do conjunto de elementos existentes. Assim, é possível buscar um elemento pela tabela com acesso direto pela Função de Espalhamento.

Podemos visualizar uma implementação do código de busca a seguir, que, além da tabela, recebe a chave de busca do elemento desejado, retornando o ponteiro do elemento caso encontrado. No caso de não encontrar o elemento, o retorno será o valor *NULL*.

```
MatAluno* busca_Esp (Hash tab, int RA) {
    int h = funcao_Esp(RA);
```

```
    while (tab[h] != NULL) {
        if (tab[h] -> RA == RA)
            return tab[h];

        h = (h + 1) % tam;
    }
    return NULL;
}
```

Verificação do tamanho do conjunto em Tabelas de Espalhamento

A função de verificação do tamanho do conjunto pode ser realizada de duas formas:

- Percorrer todas as Listas da tabela, contando os elementos todas as vezes em que a função for solicitada; ou
- Guardar em uma variável a quantidade de elementos presentes na Tabela de Espalhamento. Essa é a forma mais eficiente de ser executada.

Para a criação de uma variável, deve-se incrementá-la todas as vezes em que um elemento for adicionado ou decrementá-la todas as vezes em que um elemento for removido da Tabela de Espalhamento.

Como exemplo, podemos implementar a função para verificar o tamanho do conjunto pelo trecho de código a seguir:

```
MatAluno* tamanho_Esp (Hash tab) {
    int contador, total = 0;

    while (contador >= tam){
      if (tab[contador] != NULL){
          total++;
      }
      contador++;
    }
    return total;
}
```

Sem medo de errar

Continuando nosso desafio, sabemos que seu grupo foi destaque da faculdade em programação e vocês foram convidados a desenvolver o novo sistema da biblioteca, que está sendo reestruturada em melhorias para atender à grande demanda de alunos.

O sistema atual já não suporta mais a quantidade de buscas em razão da quantidade de exemplares cadastrados, o que o deixa mais lento, além de ser um sistema antigo.

Ao definir a forma de buscas a ser realizada pelo sistema, será necessário, agora, analisar a adição e a remoção de novos exemplares no sistema, além de verificar se existe ou não um exemplar no sistema da biblioteca, quando solicitado pelo aluno, com base em Funções de Espalhamento.

Vocês precisam pesquisar as operações possíveis de implementação no sistema, utilizando a estrutura de dados da Tabela de Espalhamento, conforme foi definida para criar o sistema. Assim, terão o desafio de desenvolver o sistema para realizar a busca de exemplares na biblioteca e apresentar um relatório da possível implementação da solução encontrada para esse desafio.

Para solucionar esse desafio, será necessária a estruturação de como segmentar os exemplares, a fim de facilitar sua busca por uma forma mais simples. Pode-se segmentar por área do conhecimento, atribuindo a cada uma delas uma letra de identificação, como:

- A = tabela[0] -> Artes
- B = tabela [1] -> Ciências Biológicas
- C = tabela [2] -> Ciências da Saúde
- D = tabela [3] -> Ciências Humanas
- E = tabela [4] -> Ciências Exatas e da Terra

e assim por diante, para definir as Tabelas de Espalhamento por segmentação.

Após definir como será realizada a segmentação, os exemplares deverão ser identificados por um código que represente a tabela à

qual pertencem, a ser inserido no sistema com base na estrutura a seguir:

```
struct exemplares{
    int codigo;
    char titulo[100];
    char autor[41];
    char area[7];
};
typedef struct exemplares Acervo;
typedef Acervo* Hash[tam];

int funcao_Esp (int codigo) {
   return (codigo % tam);
}

Acervo* insere_Esp (Hash tab, int codigo, char* titulo, char* autor, char* area) {
    int h = funcao_Esp(codigo);

    /* Criamos a alocação dinâmica de memória do tipo Acervo */
    tab[h] = (Acervo)* malloc(sizeof(Acervo));
    /* O campo código da alocação recebe o código do livro */
    tab[h] -> codigo = codigo;
   /*A função strcpy copia o texto da variável vinda pelo parâmetro para o campo do ponteiro */
    strcpy(tab[h] -> titulo, titulo);
    strcpy(tab[h] -> autor, autor);
    strcpy(tab[h] -> area, area);
    /* Retorna a tabela */
    return tab[h];
}

void remove_Esp(Hash tab, int codigo){
    int h = funcao_Esp(codigo);
```

```
    if(tab[h] -> codigo == codigo) {
        tab[h] = NULL;
        printf("\nExemplar excluido!");
     }else{
        printf("\nExemplar nao encontrado");
    }
}

Acervo* busca_Esp (Hash tab, int codigo) {
    int h = funcao_Esp(codigo);

    while (tab[h] != NULL) {
        if (tab[h] -> codigo == codigo)
            return tab[h];

        h = (h + 1) % tam;
    }
    return NULL;
}
```

Avançando na prática

Urnas eletrônicas

Descrição da situação-problema

Com o seu notório conhecimento adquirido na faculdade sobre Tabelas de Espalhamento, uma empresa de desenvolvimento de sistemas contratou seus serviços para auxiliá-la a desenvolver um sistema de criptografia de mensagens a ser utilizada dentro das novas urnas eletrônicas das próximas eleições, a fim de que toda a comunicação dentro da urna seja protegida contra algum acesso indevido ou interceptação de dados.

Ao utilizar a criptografia associada a Tabelas de Espalhamento, você precisa desenvolver um protótipo que deve receber a mensagem, retorná-la criptografada e apresentar novamente a mensagem original, como teste de que a criptografia está correta.

Ao desenvolver esse protótipo, você estará criando um sistema seguro e com garantia de máxima proteção das informações dessas eleições.

Agora é com você! Vamos implementar esse sistema?

Resolução da situação-problema

A implementação desse sistema de senha pode ser criada com base na troca de caracteres, usando a ideia do Criptograma de César, com base no vocabulário. É passada uma chave (um número) e a letra digitada é alterada pela letra que se situa na posição indicada pela chave:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>
#define tam 250

typedef struct{
      char mensagem[tam];
}urna;

void criptografa(urna *cripto);
void descriptografar(urna *cripto);
void criptografa(urna *cripto){
    printf("Informe a mensagem: ");
    gets(cripto -> mensagem);

    for(int i = 0; i < strlen(cripto -> mensagem); i++){
        if(cripto -> mensagem[i] == 'z'){
            cripto -> mensagem[i] = 'c';
        } else if(cripto -> mensagem[i] == 'y'){
                    cripto -> mensagem[i] = 'b';
```

```
                } else if (cripto -> mensagem[i] == 'x'){
                        cripto -> mensagem[i] = 'a';
                            } else {
                        cripto->mensagem[i] = cripto -> mensagem[i]
+ 3 ;
                    }
        }
        printf("\nMensagem criptografada: ");
        for(int i = 0 ; i < strlen(cripto -> mensagem); i++){
          putchar(cripto -> mensagem[i]);
        }
    }

    void descriptografar(urna *cripto){
        for(int i = 0; i < strlen(cripto -> mensagem); ++i){
            cripto -> mensagem[i] = (char)((int)cripto -> mensagem[i]
- 3);
       }

        printf("\nMensagem original: ");
        printf("%s ", cripto -> mensagem);
        printf("\n");
    }

    int main (){
        urna mensagem;
        criptografa(&mensagem);
        descriptografar(&mensagem);
        printf("\n");
        system("Pause");
        return 0;
    }
```

Faça valer a pena

1. Na Função de Espalhamento, podemos utilizar uma das formas de identificação de índice para o armazenamento na Tabela de Espalhamento, em que a função retornar um valor de índice é válida para uma das células da tabela, garantindo o acesso direto aos elementos.

Assinale a alternativa que contém a forma mais simples e utilizada de identificação por meio da Função de Espalhamento:

a) Cálculo de endereço.

b) Acesso aleatório.

c) Divisão.

d) Multiplicação.

e) Cálculo de índice.

2. Em cálculo de endereços e _____, _____ de Espalhamento, ao inserirmos um elemento na tabela e esse elemento _____ com outro elemento no endereço de _____, o elemento a ser inserido será armazenado no próximo _____ disponível na própria tabela.

Assinale a alternativa que contém as palavras que completam a sentença corretamente:

a) multiplicação – Tabelas – colidir – tabela – índice.

b) *Hash* – Funções – troca – índice – grupo.

c) divisão – Tabelas – troca – índice – grupo.

d) divisão – Funções – colidir – índice – índice.

e) *Hash* – Tabelas – colidir – tabela – índice.

3. Por meio de uma Função de Espalhamento, ao serem inseridos os elementos do vetor na lista, a função verifica a estrutura para identificar onde será inserido um novo valor, com base na Função de Espalhamento, de forma que a lista fique ordenada.

Dadas as sentenças a seguir, assinale V para verdadeiro ou F para falso:

( ) A função de Cálculo de Endereços se baseia na distribuição de valores a ordenar na estrutura.

( ) A forma de divisão é a mais simples e utilizada para a Função de Espalhamento.

( ) Para definir o endereço de um elemento na Tabela de Espalhamento, basta utilizar o resto da multiplicação de sua chave pela quantidade de elementos no vetor de alocação.

( ) Ao inserirmos um elemento na tabela e esse elemento colidir com outro elemento no endereço de índice, o elemento a ser inserido será armazenado no próximo índice disponível na própria tabela.

( ) Uma Tabela de Espalhamento sempre será totalmente preenchida, assim nunca haverá uma posição disponível para armazenamento na tabela.

Assinale a alternativa que contém a sequência correta:

a) V – F – V – V – F.

b) V – V – F – V – F.

c) F – V – F – V – F.

d) V – F – F –V – V.

e) F – V – V – F – V.

Seção 3.3

Otimização de Tabelas de Espalhamento

Diálogo aberto

Saudações, caro aluno!

Nesta seção, você poderá identificar por que ocorre a deterioração de desempenho em Tabelas de Espalhamento, podendo comprometer o desempenho na busca e no armazenamento de informações. Conhecerá como utilizar técnicas na otimização de desempenho em uma tabela, aprenderá como realizar a redução de colisões nos casos em que a colisão ocorrer em armazenamento de informações e melhorará o Espalhamento em tabelas, bem como suas aplicações na solução de problemas.

Técnicas de otimização permitem que o uso de Tabelas de Espalhamento melhore consideravelmente o desempenho de busca e armazenamento por meio do acesso direto à informação.

Para retornarmos ao nosso desafio desta unidade, a biblioteca da unidade universitária em que você estuda está passando por uma reestruturação de melhoria em diversos pontos. Como você e seu grupo se destacaram em programação, a direção da faculdade os convidou a desenvolver esse sistema.

Após vocês terem criado o sistema de buscas de livros, assim como a adição e remoção de exemplares, chegou a hora de adicionar melhorias no sistema, para evitar que se torne lento e problemas com cadastro de novos exemplares, criando, assim, uma melhor otimização por meio das Tabelas de Espalhamento.

Para completar essa tarefa, será necessário:

- realizar um levantamento de como é possível otimizar as buscas;
- evitar que o sistema se torne obsoleto;
- identificar possíveis falhas;
- propor melhorias.

A utilização de Técnicas de Otimização em Tabelas de Espalhamento permitirá ao sistema da biblioteca um melhor desempenho e ganho no tempo em sua utilização pelos colaboradores e alunos.

Com essas informações, você precisará desenvolver um relatório de como seu sistema estará pronto para melhorias futuras e como as mudanças que podem ocorrer futuramente na biblioteca já serão possíveis nos ajustes no sistema.

Pronto para mais esse desafio? Vamos lá.

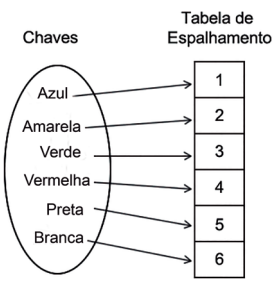
Não pode faltar

Prezado aluno, até o momento você estudou as definições de Tabelas de Espalhamento e como a Função de Espalhamento determina:

- qual o índice a ser definido para busca;
- o armazenamento de um elemento ou informação em sua estrutura.

Já estudamos que uma Função de Espalhamento Perfeita apresenta todos os elementos de um conjunto de chaves em apenas uma entrada na Tabela de Espalhamento. Em geral, chamamos a Função de Espalhamento de "perfeita" se é uma função bijetora, como mostra a Figura 3.12.

Figura 3.12 | Exemplo de Função de Espalhamento Perfeita



Fonte: elaborada pelo autor.

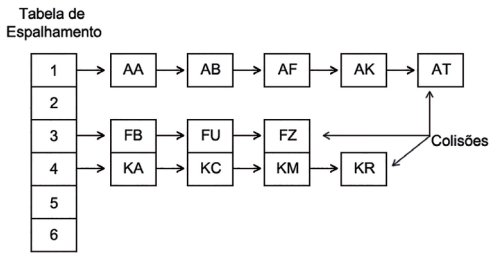
Deterioração de desempenho em Tabelas de Espalhamento

As Tabelas de Espalhamento podem sofrer com a deterioração do seu desempenho, causado principalmente pelo acúmulo de

elementos com a mesma chave calculada pela Função de Espalhamento, o que chamamos de colisão.

Segundo Celes et al. (2004), quanto mais colisões ocorrem em uma Tabela de Espalhamento, pior será o desempenho de busca ou armazenamento de informações pela Função de Espalhamento, apesar, ainda, de ser aceitável para a função, como na Figura 3.13.

Figura 3.13 | Tabela de Espalhamento com colisões aceitáveis



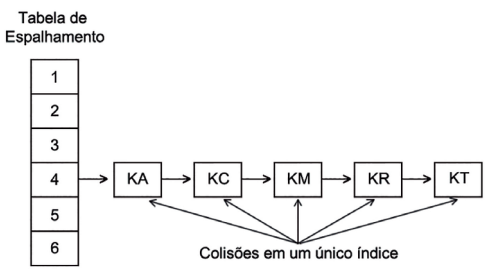
Fonte: elaborada pelo autor.

Na Figura 3.13, podemos observar que os índices 1, 2 e 3 da Tabela de Espalhamento possuem mais de um elemento armazenado em sua estrutura, necessitando de uma estrutura do tipo Lista Ligada para armazenar os elementos, enquanto os índices 2, 5 e 6 não possuem nenhum elemento armazenado.

Apesar de ser aceitável essa função, quanto mais elementos possuir um índice, maior será o esforço de busca para encontrar o elemento desejado.

Há, ainda, a situação denominada de pior caso, em que todas as chaves da Função de Espalhamento são direcionadas para um único índice da tabela, como podemos observar na Figura 3.14.

Figura 3.14 | Pior caso de uma Tabela de Espalhamento



Fonte: elaborada pelo autor.

Uma Função de Espalhamento bem definida é essencial para garantir um bom desempenho para as Tabelas de Espalhamento. Já a definição de uma função não adequada para as chaves tende a deteriorar o desempenho da tabela, como utilizar uma função sem tratamento de colisões fará a função ter um desempenho prejudicado.

Conforme Celes et al. (2004), para garantir um bom desempenho da função, como requisito básico esta deve prover uma distribuição uniforme nos índices da tabela. A não uniformidade na distribuição tende a aumentar o número de colisões, assim como o esforço para buscar ou armazenar os elementos na tabela. No entanto, algumas técnicas de otimização podem ser adotadas para garantir uma melhor distribuição dos elementos na Tabela de Espalhamento.

Assimile

A principal vantagem de ter uma Tabela de Espalhamento perfeita é o fato de ser possível realizar pesquisas com tempo constante. Assim, sendo de conhecimento todas as chaves de início, uma Tabela de Espalhamento perfeita pode ser criada por uma Função de Espalhamento Perfeita, sem ocasionar nenhuma colisão.

Técnicas de otimização em Tabelas de Espalhamento

Segundo Celes et al. (2004), uma colisão é gerada por meio da Tabela de Espalhamento, com base no cálculo do mesmo índice para duas chaves distintas. Qualquer que seja a Função de Espalhamento, existe a possibilidade de colisões que devem ser resolvidas para se obter uma distribuição de registros da forma mais uniforme possível.

O ideal seria uma Função de Espalhamento tal que dada uma chave 1 <= I <= 26, a probabilidade de a função retornar a chave X seja PROB(Fh(x) = I) = 1/26, ou seja, não tenha colisões. Tal resultado é difícil, se não impossível.

Assimile

A fórmula da Função de Espalhamento PROB(Fh(x) = I) = 1/26 indica que a probabilidade de uma função (Fh(x) seja igual ao valor do índice (I), no qual a probabilidade será de 1 em 26, sendo 26 a quantidade de letras do alfabeto.

Como você já aprendeu, para evitar que colisões ocorram no cálculo de endereço de uma chave, pela Função de Espalhamento, a única forma é saber, de início, quais as chaves possíveis para inserção, criando uma Função de Espalhamento Perfeita.

Conforme Drozdek (2016), existem diversas técnicas de otimização para funções, no entanto as três formas mais utilizadas são:

- Endereçamento Fechado;
- Endereçamento Aberto;
- Encadeamento Separado.

Segundo Drozdek (2016), no que concerne ao Endereçamento Fechado, não há mudança na posição de inserção. Os elementos devem ser inseridos sempre em uma mesma posição. Isso deve ocorrer mediante uma lista ligada, referente a cada índice da tabela. Já no caso do Endereçamento Aberto, os elementos em questão devem ser inseridos de maneira direta em cada uma das posições calculadas pela Função de Espalhamento. Caso ocorram colisões, o registro ao qual se refere a colisão deverá ser inserido na próxima posição livre da Tabela de Espalhamento.

Exemplificando

Podemos identificar esse tipo de estrutura de dados quando observamos o funcionamento de um hospital. Sempre que chega um paciente para ser internado, o hospital verifica qual o tipo de enfermidade do paciente e o direciona para uma ala. É verificada a situação dos quartos e o paciente é colocado no próximo quarto disponível.

Segundo Celes et al. (2004), o Endereçamento Aberto pode ser definido em duas formas diferentes, em:

- Função de Espalhamento Linear: procura a próxima posição vazia após o endereço-base da chave.
- Função de Espalhamento Duplo: uma Função de Espalhamento secundária calcula o incremento, substituindo o incremento da posição em 1 (um).

Já no Encadeamento Separado, em vez de indexar um registro diretamente no índice calculado com base em sua chave pela

Função de Espalhamento, o registro é indexado em uma estrutura de dados suplementar, tal como uma lista encadeada. Quando uma colisão ocorre, o registro que colide é inserido na lista encadeada.

Redução de colisões em Tabelas de Espalhamento

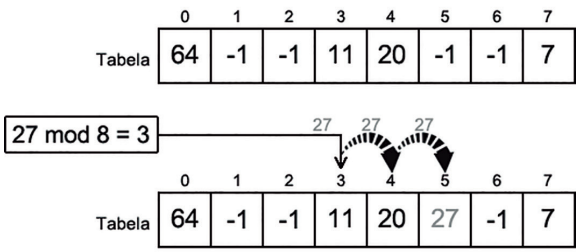
Já sabemos que o Endereçamento Aberto pode ser definido em Função do Espalhamento Linear ou da Função de Espalhamento Duplo, como podemos compreender a seguir.

Função de Espalhamento Linear

Segundo Silva (2007), a função linear, também conhecida como *Overflow* Progressivo, consiste em procurar a próxima posição vazia depois do índice-base da chave. É uma função que possui como vantagem a simplicidade em definir o índice da Tabela de Espalhamento. No entanto, sua desvantagem está no caso de ocorrerem muitas colisões, pois pode haver um agrupamento (*clustering*) de chaves em determinado índice. Isso pode tornar necessários muitos acessos para recuperar determinado elemento.

Na Figura 3.14, podemos observar o esquema de Função de Espalhamento Linear utilizando o método de divisão para encontrar o índice.

Figura 3.15 | Exemplo da Função de Espalhamento Linear



Fonte: elaborada pelo autor.

Na Tabela de Espalhamento da Figura 3.15, temos:

- os índices 0 (zero), 3 (três), 4 (quatro) e 7 (sete) já estão com elementos alocados;
- as posições sem alocação de elementos são identificadas com -1 (menos um).

Para inserir o elemento 27 na tabela:

• primeiramente, dividimos o valor do elemento (27) pela quantidade de posições da tabela (8);

• utilizamos o resto da divisão (mod) para identificar em qual índice deve ser inserido o elemento.

Como o índice 3 (três) já está ocupado, a Função de Espalhamento transfere o elemento para a próxima posição da tabela, o índice 4 (quatro). No entanto, essa posição também está ocupada, assim a função o transfere para o próximo elemento, até encontrar a próxima posição vazia na tabela; neste caso, a posição de índice 5 (cinco).

Para a implementação da Função de Espalhamento Linear, podemos utilizar um trecho do código a seguir:

```
int espalha(int chave) {
    return chave % 10;
}
int insere(int a[], int chave, int n) {
    int cont = 0, i;
    i = espalha(chave);
    /* Busca a próxima posição livre */
    while (a[ i ] != -1) {
      if (a[ i ] == chave)
          return –1; /* Elemento já existente na tabela */
      if (++cont == n)
          return –2; /* Retorna que a tabela está cheia */
    }
       /* Insere na próxima posição livre encontrada e retorna a posição onde inseriu */
     a[ i ] = chave;
     return i;
}
int busca_espalha(int a[], int chave, int n) {
```

```
    int i, cont = 0 ;
    i = espalha(chave);
    /* Procura a chave a partir da posição i */
    while (a[ i ] != chave) {
      if (a[ i ] == -1)
          return –1; /* Retorna que não achou a chave, pois existe
uma posição vazia */
      if (++cont == n)
          return –2; /* Retorna que a tabela está cheia */
    }
    /* Retorna a posição que encontrou */
    return i;
}
```

Função de Espalhamento Duplo

Conhecida também como *re-hash*, a Função de Espalhamento Duplo, em vez de incrementar a posição do elemento até a próxima posição vazia, utiliza uma Função de Espalhamento auxiliar para calcular qual o incremento que será dado à posição, levando em consideração o valor da chave.

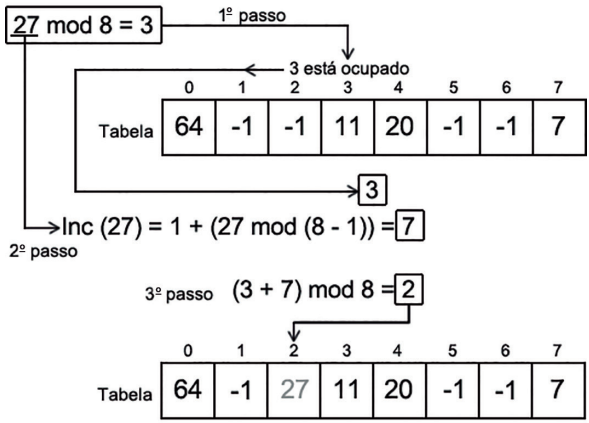
A função dupla tende a espalhar melhor ao longo dos índices da tabela, o que é uma vantagem sobre a função linear. No entanto, como desvantagem, os índices podem estar muito distantes um do outro, provocando buscas adicionais na tabela.

Ao aplicar a função dupla, se o índice estiver ocupado, será aplicada uma segunda Função de Espalhamento para obter um número X. Esse número X é adicionado ao índice gerado pela primeira Função de Espalhamento, para produzir um novo índice da tabela. Caso o novo índice também esteja ocupado, continua-se somando X ao índice da tabela, até ser encontrada uma posição vazia.

Segundo Celes et al. (2004), neste caso, para o primeiro cálculo é utilizada a função $h(k) = k \bmod N$, em que h é a função, k é a chave e N é o número de posições da tabela. Caso haja uma colisão, inicialmente calculamos $h2(k)$, que pode ser definida como $h2(k) = 1 + (k \bmod (N-1))$. Em seguida, é calculada a função

re-hashing: *rh(i,k) = (i + h2(k)) mod N*, sendo *i* o índice gerado, como podemos observar na Figura 3.16.

Figura 3.16 | Exemplo de Função de Espalhamento Duplo



Fonte: elaborada pelo autor.

Na Figura 3.16, podemos observar que o elemento 27 será inserido na Tabela de Espalhamento da seguinte forma:

• 1º passo: é realizada a divisão do elemento (27) pelo número de posições na tabela (8), resultando, no resto (mod) da divisão, o índice 3 (três), porém a posição com o índice 3 (três) já está ocupada, gerando uma colisão.

• 2º passo: a Função de Espalhamento executa a segunda função de incremento do índice com a função *1 + (k mod (N-1))*, resultando em 7 como o resto da divisão.

• 3º passo: é executada a função *re-hash*, com a soma dos índices gerados nos dois cálculos, do 1º passo (3) e 2º passo (7), dividinda-a a pelo número de posições da tabela (8) e resultando no índice 2 (dois). Caso essa posição esteja vazia, o elemento 27 será adicionado nela; caso contrário, serão executados novamente os passos 2 e 3.

Podemos utilizar a implementação da Função de Espalhamento Duplo com base no trecho de código a seguir:

```
int espalha(int chave, int N) {
    return (chave % N); /* Calcula o índice da tabela */
}
```

```
    int espalha2(int chave, int N) {
        return (1 + (chave % (N-1))); /* Calcula o segundo índice da
tabela */
    }
    int insere(item a[ ], item chave, int N) {
        int i = espalha(chave, N); /* Variável i recebe retorna da função
*/
        int k = espalha2(chave, N); /* Variável k recebe retorna da
2ª função */
        int cont = 0;
        /* Busca a próxima posição livre */
        while (a[ i ] != -1) {
            if (a[ i ] == chave)
                return –1; /* Retorna se o elemento já existe na tabela */
            if (++cont == N)
                return –2; /* Retorna que a tabela está cheia */
            i = ((i + k) % N); /* Calcula a função re-hash */
        }
        /* Insere na próxima posição livre encontrada */
        a[ i ] = chave;
        return i;
    }
    int busca_espalha (item a[ ], item chave, int N) {
        int i = espalha(chave, N); /* Variável i recebe retorna da função
*/
        int k = espalha2(chave, N); /* Variável k recebe retorna da
2ª função */
        int cont = 0 ;
        /* Procura a chave a partir da posição i */
        while (a[ i ] != chave) {
            if (a[ i ] == -1)
```

```
            return –1; /* Retorna que não achou a chave, pois existe uma posição vazia */
        if (++cont == n)
            return –2; /* Retorna que a tabela está cheia */
        i = (i + k) % N; /* Calcula a função re-hash */

    }
    /* Retorna a posição que encontrou */
    return i;
}
```

Reflita

As chaves utilizadas em uma Tabela de Espalhamento são uma identificação do elemento que desejamos inserir. Normalmente, as chaves são identificadas como números. Poderíamos utilizar *strings* como identificadores de chaves?

Quais as vantagens e desvantagens de utilizar mais ou menos caracteres na Função de Espalhamento para o cálculo do índice?

Melhorando o Espalhamento

Você já estudou que uma Tabela de Espalhamento pode ter o desempenho comprometido à medida que vai ficando cheia. Enquanto possui poucos elementos, a busca é muito eficiente e rápida para qualquer tipo de Função de Espalhamento adotada.

O desempenho da tabela diminui pela falta de posições disponíveis, ocasionando lentidão. Uma solução seria necessária para aumentar a tabela e reinserir todos os elementos novamente, contudo uma operação ineficiente a ser executada.

Conforme Drozdek (2016), para melhorar o Espalhamento, uma solução seria utilizar tabelas dinâmicas na estrutura, em vez de uma Lista Ligada para armazenar os valores. Para tal, pode ser utilizada tanto com a Função de Espalhamento Linear como com a Função de Espalhamento Duplo.

Assim, quando utilizamos esse método, no momento em que a tabela passa de N/2 elementos, dobramos o seu tamanho, fazendo com que sempre tenha menos da metade dos elementos ocupados.

O trecho de código a seguir apresenta a implementação da tabela dinâmica na Função de Espalhamento:

```
int insere_nova(item chave) {
    int i = espalha(chave, NN);
    /* Busca a próxima posição livre */
    /* Com a tabela duplicada, sempre haverá posição disponível
*/
    while (q[ i ] != VAZIO) {
        i = (i+1) % NN; /* Calcula o índice */
    }
    /* Insere na posição livre encontrada */
    q[ i ] = chave;
    return i;
}
void expande() {
    int i, NN = 2*N;
    q = malloc(NN*sizeof(item));
    limpa(q, NN);
    /* Insere todos os elementos na nova tabela */
    for (i = 0; i < N; i++)
        if (p[ i ] != VAZIO)
            insere_nova(p[ i ]);
    N = NN;
    free(p);
    p = q;
}
```

```
void insere(item chave) {
    int i;
     //Verifica se o tamanho da tabela é menor que a soma da quantidade de elementos da tabela
    if (N < M+M)
        expande(); /* Chama a função para expandir a tabela */
    i = espalha(chave, N);
    /* Busca a próxima posição livre */
    while (p[ i ] != VAZIO) {
        if (p[ i ] == chave)
            return –1; /* Retorna se o elemento já existe na tabela */
        i = (i+1) % N;  /* Calcula o índice */
    }
    /* Insere na posição livre encontrada */
    p[ i ] = chave;
    M++;
    return i;
}
```

Pesquise mais

Para reduzir colisões e melhorar o Espalhamento em Tabelas de Espalhamento, podemos realizar as inserções e buscas com o tratamento das colisões. No vídeo indicado a seguir, é apresentado como realizar esse tratamento e as funções nas tabelas.

LINGUAGEM C – Programação descomplicada. **[ED] Aula 98 - Tabela Hash - Inserção e busca com tratamento de colisão**. Disponível em:

<https://www.youtube.com/watch?v=Dhbgy2q0h4w>. 2015. Acesso em: 29 jan. 2018. (Vídeo do YouTube.)

A vantagem de uso da Função de Espalhamento Linear está no acesso mais rápido, enquanto a Função de Espalhamento Duplo utiliza melhor a memória disponível, mas ambas dependem do tamanho da memória disponível que permitirá a expansão da tabela.

Sem medo de errar

Sabendo que seu desafio parte da biblioteca da unidade universitária em que estuda, a qual está passando por uma reestruturação de melhoria em diversos pontos, você e seu grupo se destacaram em programação na unidade e, por esse motivo, a direção da faculdade convidou-os a desenvolver esse sistema.

Após terem criado o sistema de buscas de livros, assim como adição e remoção de exemplares, chegou a hora de adicionar melhorias no sistema, para evitar que o sistema se torne lento e problemas com cadastro de novos exemplares, criando, assim, uma melhor otimização do sistema por meio das Tabelas de Espalhamento.

Para completar essa tarefa, será necessário:

- realizar um levantamento de como é possível otimizar as buscas;
- desobrir como evitar que o sistema se torne obsoleto;
- identificar possíveis falhas;
- propor melhorias.

A utilização de Técnicas de Otimização em Tabelas de Espalhamento permitirá ao sistema da biblioteca um melhor desempenho e ganho no tempo em sua utilização pelos colaboradores e alunos.

Com essas informações, desenvolva um relatório de como seu sistema estará pronto para melhorias futuras e como as mudanças que podem ocorrer futuramente na biblioteca já serão passíveis de ajustes no sistema.

Como resolução desta atividade, você precisa compreender e aprender as técnicas de Espalhamento existentes e definir qual delas melhor se adapta ao sistema em desenvolvimento.

Pode-se implementar a técnica de dobramento de tabela com o uso de tabelas dinâmicas, se considerarmos a grande quantidade de exemplares a serem cadastrados.

Uma das técnicas de otimização que pode ser utilizada é a de Espalhamento Aberto, com o método de Função de Espalhamento Duplo, que permitirá que todo livro cadastrado no sistema tenha o endereço calculado de forma a evitar colisões e também a utilização do trecho de código a seguir, a ser implementado no sistema:

```
int espalha(int chave, int N) {
    return (chave % N); /* Calcula o índice da tabela */
}

int espalha2(int chave, int N) {
     return (1 + (chave % (N-1))); /* Calcula o segundo índice da tabela */
}

int insere(item a[ ], item chave, int N) {
    int i = espalha(chave, N); /* Variável i recebe retorna da função */
     int k = espalha2(chave, N); /* Variável k recebe retorna da 2ª função */
    int cont = 0;

    /* Busca a próxima posição livre */
    while (a[ i ] != -1) {
        if (a[ i ] == chave)
            return –1; /* Retorna se o elemento já existe na tabela */
        if (++cont == N)
             return –2; /* Retorna que a tabela está cheia */
        i = ((i + k) % N); /* Calcula a função re-hash */
    }

    /* Insere na próxima posição livre encontrada */
    a[ i ] = chave;
    return i;
}
```

```
int busca_espalha (item a[ ], item chave, int N) {
    int i = espalha(chave, N); /* Variável i recebe retorna da função */
     int k = espalha2(chave, N); /* Variável k recebe retorna da 2ª função */
    int cont = 0 ;

    /* Procura a chave a partir da posição i */
    while (a[ i ] != chave) {
        if (a[ i ] == -1)
            return –1; /* Retorna que não achou a chave, pois existe uma posição vazia */
        if (++cont == n)
            return –2; /* Retorna que a tabela está cheia */

        i = (i + k) % N; /* Calcula a função re-hash */

    }

    /* Retorna a posição que encontrou */
    return i;
}
```

Avançando na prática

Atendimento ao cliente

Descrição da situação-problema

Uma nova demanda apareceu e você está prestes a trabalhar para uma grande empresa do país. Uma empresa de celular precisa melhorar seu sistema para torná-lo o mais ágil possível no atendimento e na identificação dos clientes que realizam chamados para um atendimento mais personalizado.

Sua função é planejar e definir como será a Função de Espalhamento do sistema e como este deverá funcionar para ter o melhor desempenho possível em suas buscas.

Você precisa entregar um relatório informando o planejamento e como será o funcionamento desse sistema.

Vamos começar?

Resolução da situação-problema

Após as análises no funcionamento da empresa e em seu atual sistema, você identificou que a utilização de uma estrutura de dados do tipo Tabela de Espalhamento, com Função de Espalhamento Duplo, terá melhor adaptação ao tipo de sistema para obter melhor desempenho.

Tabelas dinâmicas também são úteis se pensarmos em sua utilização, pois dobrarão o tamanho da tabela dinamicamente, permitindo melhor busca de informações nela, considerando, ainda, que a utilização de função dupla, com tabela dinâmica, fará o sistema aproveitar melhor o uso da memória.

O sistema deve buscar os dados de um cliente somente com a passagem do número de telefone no sistema, e este deve retornar as informações de forma rápida, acessando diretamente a tabela, como o exemplo da Figura 3.17.

Figura 3.17 | Acesso direto ao nome do cliente por meio do telefone

Vazio	Vazio	Vazio	José da Silva	Vazio
99635-8902	99635-8903	99635-8904	99635-8905	99635-8906

Fonte: elaborada pelo autor.

Assim, a utilização do número de telefone como chave permite uma consulta mais rápida à posição das informações do cliente na tabela.

Faça valer a pena

1. Uma Função de Espalhamento bem definida é essencial para garantir um bom desempenho para as Tabelas de Espalhamento. Já a definição de uma função não adequada para as chaves tende a deteriorar o desempenho da tabela, permitindo que o acesso direto aos itens seja mais demorado que o normal.

Assinale a alternativa que indica qual situação garante que a Função de Espalhamento seja considerada perfeita:

a) Todos os elementos devem estar obrigatoriamente ocupando todas as posições da tabela.

b) Nas Tabelas de Espalhamento, a colisão ocorrerá somente com chaves numéricas.

c) Onde todos os elementos da chave possam estar distribuídos pela tabela separados por índices.

d) Todos os elementos da chave possuem o mesmo índice da tabela.

e) As Tabelas de Espalhamento precisam estar sem nenhum elemento para serem perfeitas.

2. Uma colisão é gerada por meio da Tabela de Espalhamento com base no cálculo do mesmo índice para duas chaves distintas. Qualquer que seja a Função de Espalhamento, há a possibilidade de colisões, que devem ser resolvidas para obter uma distribuição de registros da forma mais uniforme possível.

Considerando as possíveis colisões na tabela, assinale a alternativa que apresenta as formas de otimização de função para colisões:

a) Encadeamento Separado, Endereçamento Aberto e Endereçamento Fechado.

b) Endereçamento Separado, Endereçamento Aberto e Endereçamento Fechado.

c) Endereçamento Separado, Encadeamento Aberto e Encadeamento Fechado.

d) Encadeamento Separado, Encadeamento Aberto e Encadeamento Fechado.

e) Endereçamento Separado, Endereçamento Aberto e Endereçamento Fechado.

3. O desempenho da tabela diminui pela falta de posições disponíveis, ocasionando lentidão. Uma solução seria necessária para aumentar a tabela e reinserir todos os elementos novamente, o que é uma operação ineficiente a ser executada.

Assinale a alternativa que possui a técnica de melhora do Espalhamento:

a) Utilizar tabelas estáticas e chaves fixas.

b) Utilizar tabelas dinâmicas e chaves fixas.

c) Utilizar tabelas dinâmicas e deixar o tamanho da tabela indefinido.

d) Utilizar tabelas dinâmicas e dobrar o tamanho da tabela.

e) Utilizar tabelas estáticas e dobrar o tamanho da tabela.

Referências

CELES, W.; CERQUEIRA, R.; RANGEL, J. S. **Introdução à estrutura de dados**: com técnicas de programação em C. Rio de Janeiro: Campus Elsevier, 2004.

DROZDEK, A. **Estrutura de dados e algoritmos em C++**. Tradução da 4. ed. norte-americana. São Paulo: Cengage Learning, 2016.

SILVA, O. Q. **Estrutura de dados e algoritmos usando C**: fundamentos e aplicações. Rio de Janeiro: Ciência Moderna, 2007.

Armazenamento associativo

Convite ao estudo

Prezado aluno, estamos chegando à última unidade de estudos sobre Algoritmos e Estrutura de dados. Na unidade anterior, você aprendeu acerca das Tabelas de espalhamento, seu funcionamento, sua estrutura, como otimizar o acesso aos dados armazenados em sua estrutura.

Para concluir nossos estudos, nesta unidade você vai conhecer e aprender sobre Armazenamento Associativo, sua definição, motivação, para poder utilizar essa estrutura e seu funcionamento com as técnicas de listas e espalhamento, aprimorando, assim, seus conhecimentos em Algoritmos e Estrutura de dados.

Você conhecerá e compreenderá as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas, e também os Mapas de Armazenamento Associativo, sua construção e uso adequados e sua aplicação em programas de computador.

Nesta primeira seção, você compreenderá a definição de Armazenamento Associativo, quais as motivações para utilizar essa estrutura, como funcionam os Mapas Associativos e usos gerais para Armazenamento Associativo.

Na seção seguinte, você aprenderá sobre definição e exemplos de Mapas com Listas, verificará a existência de uma Chave específica em um mapeamento com listas, como adicionar e remover associações dadas suas chaves em mapeamento com listas e recuperar valores associados a chaves em mapeamento com listas.

Para finalizar nosso conteúdo sobre Armazenamento Associativo e em Algoritmos e Estrutura de dados, você conhecerá a definição e exemplos de Mapas com

Espalhamento, verificará se existe uma Chave específica em Mapas com Espalhamento, poderá adicionar ou remover associações dadas suas chaves em Mapas com Espalhamento e recuperará valores associados a chaves em Mapas com Espalhamento.

Como desafio desta unidade, será proposto o seguinte contexto prático: a sra. Fátima, uma cozinheira, decidiu abrir seu primeiro restaurante e se tornar uma empresária do ramo da culinária. Ela contratou a sua empresa, que possui um setor de desenvolvimento de *softwares*, para implantar um sistema para o restaurante dela. Esse sistema será fundamental para a sra. Fátima, pois, por meio dele, ela controlará as vendas e a montagem dos cardápios, assim como controlará o estoque dos ingredientes dos pratos.

Esse será um desafio para você, pois precisará utilizar mais conhecimento em Armazenamento Associativo, mapeamento com listagem de produtos e mapeamento de espalhamento para realizar o controle dos pratos. À medida que é vendido um prato, é dada a baixa no estoque de todas as matérias-primas que o compõem.

Agora chegou a hora de você adquirir novos conhecimentos e colocar em prática todos eles a fim de solucionar esse desafio.

Você consegue auxiliar a sra. Fátima a abrir o restaurante dela?

Vamos começar ampliando nossos conhecimentos em Armazenamento Associativo com esta unidade para aplicá-los em resoluções de problemas do mundo real.

Prontos?

Seção 4.1

Definição e usos de Mapas de Armazenamento

Diálogo aberto

Caro aluno, na unidade anterior, você conheceu e aprendeu a respeito das Tabelas de espalhamento, definições, operações, como resolver problemas e otimizar as pesquisas de informações nessa estrutura de dados. Nesta seção, vamos direcionar nossos estudos em Armazenamentos Associativos com o objetivo de conhecer e compreender os Mapas de Armazenamento Associativo, sua construção e uso adequados, e sua aplicação em programas de computador.

As estruturas de Armazenamento Associativo permitem armazenar elementos em estruturas como Listas ou Mapas com Espalhamento sem depender da posição onde estão armazenados, baseando-se apenas em seu valor. Podemos citar como exemplo a compra de uma passagem de avião. Ao comprar a passagem, você recebe um número localizador, que é sua chave de identificação, e, com ela, estão associados seus dados pessoais, sua poltrona, sua bagagem e seu destino. Todas essas informações associadas são independentes de quem comprou antes ou depois de você.

Nesta seção, você vai conhecer e compreender as definições, as motivações, o que são Mapas associativos e usos gerais de uma estrutura de Armazenamento Associativo.

Para iniciarmos nosso estudo sobre Armazenamento Associativo, vamos ao nosso desafio desta seção.

Como é de seu conhecimento, a sra. Fátima contratou você para desenvolver o sistema do restaurante para controlar as vendas, a montagem dos cardápios, assim como o estoque dos ingredientes utilizados no restaurante.

Para início do nosso desafio profissional, a sra. Fátima deseja que, no cardápio, os pratos sejam identificados por número. A ideia

é a mesma que as grandes redes de *fast food* utilizam, em que, solicitando o prato pelo número, seja dada a baixa nos ingredientes do prato.

Vamos utilizar a estrutura de dados de Armazenamento Associativo visto nesta seção de como funciona e como sua finalidade pode ser usada para encontrar uma solução a esse desafio.

Será necessário que você realize uma pesquisa e levantamento de como deve ser a forma de realizar a associação entre o número e os ingredientes do prato?

Você vai precisar entregar um relatório detalhado a sra. Fátima, documentando como será o funcionamento do sistema.

Os assuntos desta seção são de muita importância para seu crescimento profissional e formação, devendo realizar as atividades propostas e acompanhar todo o contexto pedagógico com muita atenção.

Vamos começar?

Não pode faltar

Prezado aluno, nesta seção você vai aprender sobre Armazenamento Associativo, uma estrutura que permite criarmos um mapeamento entre o elemento que desejamos e uma chave para identificá-lo.

Segundo Tenenbaum (2007), uma estrutura de dados precisa dispor de implementações para gerenciar e controlar como o armazenamento será atribuído a uma linguagem de programação.

Definição de Armazenamento Associativo

Segundo Goodrich e Tamassia (2013), o Armazenamento Associativo é uma estrutura que permite o acesso aos seus elementos com base apenas no seu valor, independentemente de sua posição na estrutura. Em determinados casos, é utilizado apenas parte do valor do elemento, conhecido como chave, em vez do valor do elemento completo.

Um Armazenamento Associativo tem como base o tipo de estrutura de tabelas na construção e como participação fundamental para o desenvolvimento de sistemas. Uma das principais aplicações é na construção de tabelas de símbolos, com plena utilização em compiladores e montadores.

As tabelas também são utilizadas em sistemas operacionais, como tabelas de arquivos e tabelas de processos para a gestão de informações necessárias à execução de programas.

Na Figura 4.1, podemos visualizar um exemplo de tabela associativa de Código Morse, onde um caractere representa uma sequência de sinais curtos ou longos.

Figura 4.1 | Tabela com Código Morse

A •–	J •–––	S •••	2 ••–––
B –•••	K –•–	T –	3 •••––
C –•–•	L •–••	U ••–	4 ••••–
D –••	M ––	V •••–	5 •••••
E •	N –•	W •––	6 –••••
F ••–•	O –––	X –••–	7 ––•••
G ––•	P •––•	Y –•––	8 –––••
H ••••	Q ––•–	Z ––••	9 ––––•
I ••	R •–•	1 •––––	0 –––––

Fonte: adaptada de: <http://brasilescola.uol.com.br/geografia/codigo-morse.htm>. Acesso em: 11 dez. 2017.

Exemplificando

Podemos exemplificar a utilização de um Armazenamento Associativo recordando a lista de chamada de uma classe do Ensino Médio ou Fundamental, quando a lista era montada em ordem alfabética e existia um número para identificar cada aluno dessa lista. Com um número associado a um nome de aluno, temos uma tabela de associação.

A utilização de código de barras em produtos industrializados é uma forma de criar associações entre o produto, o fabricante e a empresa que vende determinado produto. Cada produto possui uma identificação com base na sequência de números definidos pelo fabricante. Assim, por meio do código de barras, esse produto tem todas as informações dentro da fábrica e no revendedor é

cadastrado com as informações necessárias para sua venda. É por meio desse código que o sistema define sua entrada nas empresas, e sua saída delas, como mostra a Figura 4.2.

Figura 4.2 | Exemplo de código de barras

Fonte: <https://www.istockphoto.com/br/foto/mulher-de-varredura-de-c%C3%B3digo-de-barras-de-um-r%C3%B3tulo-no-armaz%C3%A9m-moderno-gm697462150-129182251>. Acesso em: 17 jan. 2018.

Motivação para Armazenamento Associativo

Para Goodrich e Tamassia (2013), a utilização de uma chave em mapa permite armazenar e localizar elementos que possam ser encontrados rapidamente. A motivação para as pesquisas é referente a cada elemento que armazena informações adicionais úteis junto à chave, mas que podem ser acessadas somente pela chave.

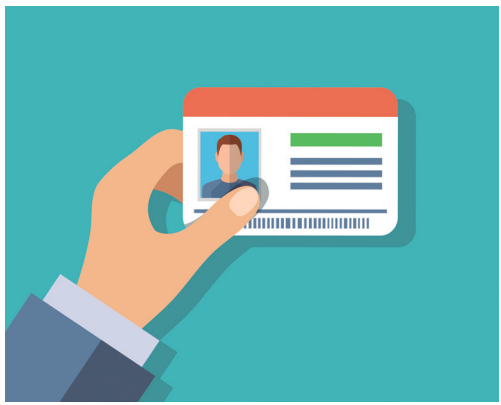
Você consegue imaginar quantas utilizações de associação nos dias de hoje são possíveis no mundo real? Pois bem, inúmeras foram as formas de utilização da associação no mundo real, como o controle de veículos existentes, como carros, motos, caminhões e ônibus nas ruas das cidades ou estradas; algum órgão governamental associa uma placa a cada veículo existente, assim é informada a placa do veículo apenas, sendo possível consultar suas informações cadastrais.

A formação da placa é a chave de identificação de um veículo.

Imagine se cada pessoa fosse identificada apenas pelo nome, como funcionariam os sistemas? E nos casos de nomes iguais? Para isso, foi implantado o CPF (Cadastro de Pessoa Física), que permite identificar uma pessoa apenas com um código de 11 (onze) dígitos. Além do

CPF, uma pessoa é identificada por diversos outros documentos, como número de CNH (Carteira Nacional de Habilitação), número de Título de Eleitor e até mesmo pelo número de RG (Registro Geral), como mostra a Figura 4.3.

Figura 4.3 | Identificação pessoal



Fonte: <https://www.istockphoto.com/br/vetor/id-cards-in-hand-gm613048018-105736117>. Acesso em: 12 dez. 2017.

Também podemos considerar o uso de CDs de músicas, compostos de diversas músicas de um cantor, grupo ou coletânea. Todas as músicas estão dentro do CD, e cada uma é identificada por uma faixa, e cada faixa, por sua vez, é associada a uma música. Você escolhe a faixa que deseja e terá acesso às informações daquela música em questão.

Podemos encontrar esse tipo de associação em um comércio. Por exemplo, o número de venda da Nota Fiscal está associado a uma lista de produtos comprados por um cliente. Assim, para saber quais produtos foram adquiridos, o vendedor digita o número da Nota e são apresentados não somente os produtos, mas também qual cliente está associado àquela compra, quais as taxas e impostos a serem pagos e outras informações para o despacho dos produtos e para o governo.

Há diversos outros exemplos em que um elemento está associado a informações adicionais ou outro elemento. Esse é o intuito de você estudar esse tipo de estrutura associativa para o desenvolvimento de diversos sistemas.

Mapas Associativos

Segundo Pereira (2008), um mapa não pode ser definido por fórmulas, diferentemente de como acontece com funções. Um mapeamento é definido por meio de uma relação entre seus pares, que nem sempre é baseada em questões lógicas ou matemáticas.

Conforme Pereira (2008), a função de Mapas Associativos é efetuar a associação entre elementos dentro da estrutura, realizando a associação entre uma chave e um valor recebidos, permitindo a recuperação rápida de um valor associado a uma chave.

Os Mapas Associativos são estruturas de dados que permitem implementar as seguintes funcionalidades:

- Adicionar uma associação.
- Verificar um valor de uma chave específica.
- Remover uma associação de uma chave específica.
- Verificar se existe uma associação para determinada chave.
- Informar a quantidade de associações na estrutura.

Para adicionarmos uma associação, podemos realizar a implementação do trecho de código a seguir, como exemplo:

```
#include <iostream>  /* Utiliza a biblioteca iostream da linguagem C++ */
#include <map> /* Utiliza a biblioteca map da linguagem C++ */

using namespace std; /* Este comando serve para definir um espaço de nome para evitar duplicidade */

int main (){
        map <int, string > mapa;  /* Definimos a estrutura do mapa com associação entre um número inteiro e uma string */
        mapa[1] = "KLS"; /* Realizamos a associação entre o número 1 e a palavra "KLS" */
        cout << mapa[1] << endl; /* Comando para imprimir na tela com a biblioteca iostream o mapeamento associativo da chave 1*/

        return 0;
}
```

Nesse trecho de código, a estrutura do mapa implementado entre um valor inteiro e uma *string* (palavra) recebe uma chave, no caso o número 1, e retorna o valor "KLS", que foi associado a ele no programa.

Pesquise mais

A utilização de Mapas Associativos tem grande aplicação dentro do sistema operacional em gerenciamento de memórias cache.

Aproveitando o estudo em Mapas Associativos, sugiro que você pesquise como funciona o gerenciamento da memória cache com base nessa aplicação, quais as vantagens dessa forma de gerenciamento e quais as formas de aplicação de Mapas Associativos em memória. Disponível em: <https://www.youtube.com/watch?v=kNFriY-nYDc>. Acesso em: 18 jan. 2018.

Em Mapas Associativos, é possível realizar a remoção, quando necessária, de uma associação dentro do mapeamento criado. Nesse caso, será preciso implementar a remoção da associação informando a chave que se deseja remover e o valor será removido do mapeamento, desfazendo-se, assim, a associação existente no mapeamento.

Para a remoção de um Mapa Associativo, implementamos uma linha especificando a chave que deve ter sua associação removida, como podemos visualizar no exemplo a seguir:

```
#include <iostream>
#include <map>

using namespace std;

int main (){
    map <int, string> mapa;

    mapa[1] = "KLS";
    cout << mapa[1] << endl;
```

```
        mapa.erase(1);  /* Chamamos o mapa criado, seguido da função erase (apagar) */

        cout << mapa[1] << endl;   /* Imprimimos novamente para verificar se foi removida a associação */

        return 0;

    }
```

Reflita

Ao utilizarmos a estrutura de mapa (*map*), estamos implementando uma estrutura definida para uso de associação entre chaves e valores. Seria possível implementar essa solução sem a necessidade de utilizar mapas (*map*)?

É possível, por meio do Mapa Associativo, verificar se determinada chave está ou não associada a algum valor, como na implementação do trecho de código a seguir:

```
#include <iostream>
#include <map>

using namespace std;

int main (){
        int chave;
        map <int, string> mapa;

        mapa[1] = "KLS";
        mapa[2] = "KROTON";
        mapa[3] = "ESTRUTURA DE DADOS";

        printf("Digite a chave: \n");
        scanf("%d",&chave);

        if(mapa.find(chave) == mapa.end())  /* Verifica se a chave existe na estrutura */
```

```
            cout << "\nChave NAO existe!\n\n"; /* Caso negativo informa que não existe */
        else
            cout << "\nChave existe! - Valor: " + mapa[chave] + "\n\n";   /* Caso afirmativo é informado que existem a chave e o valor dela */

        return 0;
    }
```

Além disso, podemos saber quantas associações existem na estrutura criada e, assim, identificar o tamanho do nosso Mapa Associativo criado com a seguinte implementação de código:

```
#include <iostream>
#include <map>

using namespace std;

int main (){
    int chave;
    map <int, string> mapa;

    mapa[1] = "KLS";
    mapa[2] = "KROTON";
    mapa[3] = "ESTRUTURA DE DADOS";

    cout << "Tamanho da estrutura de Mapa: " << mapa.size() << endl; /* Função size retorna o tamanho do mapeamento */

    return 0;
}
```

Assimile

Uma questão fundamental para o uso de Mapa Associativo é que as chaves devem ser únicas, não podendo existir dois alunos com o mesmo número de chamada, duas Notas Fiscais com o mesmo número ou duas pessoas com o mesmo número de documento CPF.

Usos gerais para Armazenamento Associativo

Conforme Goodrich e Tamassia (2013), a utilização de estruturas de Armazenamento Associativo está presente em todos os tipos de sistemas, seja um simples sistema de cadastro de clientes de uma pequena loja, seja um sistema de grande porte de uma empresa multinacional.

Qualquer sistema que tenha um cadastro é passível de Armazenamento Associativo. Em um sistema em que é necessário o cadastro de um cliente, seus dados são associados a um código para ser utilizado em uma busca para venda, ou um relatório de histórico de pagamento, e assim podemos visualizar todas as informações necessárias.

Em um sistema de supermercado, o código de barras é a chave de identificação de todas as informações dos produtos e, por meio do leitor de código de barras, temos as informações, no caixa, da descrição do produto, valor de venda e valor de imposto.

A utilização de Armazenamento Associativo pode ser aplicada também em um sistema bancário. Todos os clientes são identificados (chave) por um número de agência e número de conta. Por meio desses números, é possível associar todas as informações de um cliente, como dados pessoais, documentos e endereços, como também transações bancárias realizadas, valores em conta, cartões associados e qualquer outro tipo de informações pertencentes ao banco.

Sem medo de errar

Com o intuito de aprofundar nosso conhecimento, você foi contratado pela sra. Fátima para desenvolver o sistema do restaurante a fim de controlar as vendas, montagem dos cardápios, assim como o estoque dos ingredientes utilizados no restaurante.

Para início do nosso desafio profissional, a sra. Fátima deseja que, no cardápio, os pratos sejam identificados por número. A ideia é a mesma que as grandes redes de *fast food* utilizam, em que, solicitando o prato pelo número, seja dada a baixa nos ingredientes do prato.

Utilizando a estrutura de dados de armazenamento associativo visto nessa seção, seu objetivo é compreender seu funcionamento e sua finalidade para encontrar uma solução a esse desafio.

Será necessário que você realize uma pesquisa e levantamento de como deve ser a forma de realizar a associação entre o número e os ingredientes do prato.

Você vai precisar entregar um relatório detalhado a sra. Fátima, documentando como será o funcionamento do sistema.

Para a resolução desse desafio, é muito importante compreender o Armazenamento Associativo e como funciona o Mapa Associativo.

Podemos realizar a associação dos pratos com base em uma tabela para compreender e visualizar melhor os pratos a serem servidos.

Número	Prato
101	Parmegiana de frango
102	Parmegiana de carne
103	Almôndegas
104	Bisteca suína
105	Bife acebolado
106	Frango grelhado

Ao concluir a montagem da tabela de pratos que estarão disponíveis de início no restaurante, podemos criar a associação em forma de algoritmo para ser implementada no sistema:

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

using namespace std;

int main (){
map <int, string > pratos;
```

```
pratos[101] = " Parmegiana de Frango";
pratos[102] = " Parmegiana de Carne";
pratos[103] = " Almondegas";
pratos[104] = " Bisteca Suína";
pratos[105] = " Bife Acebolado";
pratos[106] = " Frango Grelhado";

printf ("Pratos:\n\n");

cout << pratos[101] << endl;
cout << pratos[102] << endl;
cout << pratos[103] << endl;
cout << pratos[104] << endl;
cout << pratos[105] << endl;
cout << pratos[106] << endl;

return 0;
}
```

Dessa forma, pode ser implementado o sistema para informação dos pratos disponíveis no restaurante.

Avançando na prática

Jukebox - Aparelho eletrônico de som

Descrição da situação-problema

O sr. Afonso, proprietário de uma empresa que fabrica dispositivos de som, como rádio para carros e aparelhos de som, decidiu investir em algo inovador para sua empresa.

Ele tem o projeto de começar a trabalhar com Jukebox digital, um aparelho eletrônico cuja função é tocar as músicas que estão em seu catálogo e escolhidas por clientes, geralmente sendo acionado por moedas ou dinheiro.

Como você trabalha com serviços de consultoria para empresas na área de sistemas, o sr. Afonso solicitou seus serviços para ajudá-lo a desenvolver um sistema para seu novo aparelho. Ele deseja que ao colocar a moeda ou dinheiro, o aparelho solicite ao cliente para escolher o gênero musical e, depois, optar entre 1 (um) e 5 (cinco) para ouvir as músicas da banda desejada.

Sua tarefa será criar a programação inicial para dois gêneros musicais, com 5 (cinco) bandas de cada gênero para o cliente selecionar como um protótipo inicial do projeto. Ao selecionar um gênero, o sistema deve apresentar as bandas daquele gênero disponíveis. Com a banda selecionada, o sistema apresenta uma mensagem que a música está tocando. Caso o cliente selecione um gênero ou banda fora da faixa disponível, o sistema retornará a mensagem de "Opção inválida". Você deverá entregar um relatório com a implementação.

Mãos à obra!

Resolução da situação-problema

Para resolver esta tarefa, será necessário você desenvolver o código do protótipo da Jukebox, pensando na utilização do cliente, e ter um nível de condição dentro do sistema para a escolha do gênero musical:

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

using namespace std;

int main (){
    int opcao = 0, banda;
    map <int, string > genero1;
    map <int, string > genero2;
```

```
    genero1[1] = " Skank";
genero1[2] = " Barao Vermelho";
genero1[3] = " Capital Inicial";
genero1[4] = " Titas";
genero1[5] = " Nenhum de Nos";

genero2[1] = " Leandro e Leonardo";
genero2[2] = " Gian e Geovani";
genero2[3] = " Chitaozinho e Xororo";
genero2[4] = " Jorge e Mateus";
genero2[5] = " Bruno e Marrone";

do {
banda = 0;
system("cls");
printf("Escolha o Genero:\n");
printf("1 - Rock Nacional\n");
printf("2 - Sertanejo\n");
printf("3 - Sair\n");
scanf("%d", &opcao);

switch (opcao){
            case 1:
                    system("cls");
printf("Escolha a Banda de Rock Nacional:\n");
printf("1 - Skank\n");
printf("2 - Barao Vermelho\n");
printf("3 - Capital Inicial\n");
printf("4 - Titas\n");
printf("5 - Nenhum de Nos\n");
scanf("%d", &banda);
```

```
if(genero1.find(banda) == genero1.end())
cout << "\nOpcao Invalida!\n\n";
else
cout << "\nTocando: " + genero1[banda] + "\n\n";

system("pause");
                    opcao = 0;
                    break;

                case 2:
system("cls");
printf("Escolha a Dupla Sertaneja:\n");
printf("1 - Leandro e Leonardo\n");
printf("2 - Gian e Geovanni\n");
printf("3 - Chitaozinho e Xororo\n");
printf("4 - Jorge e Mateus\n");
printf("5 - Bruno e Marrone\n");
scanf("%d", &banda);

if(genero2.find(banda) == genero2.end())
cout << "\nOpcao Invalida!\n\n";
else
cout << "\nTocando: " + genero2[banda] + "\n\n";

system("pause");
                    opcao = 0;
                    break;
            default:
                    break;
        }
    }while (opcao != 3);
        system("pause");
    return 0;
}
```

Faça valer a pena

1. Um _____ tem como base o tipo de estrutura de _____ na construção e como participação fundamental para o desenvolvimento de sistemas. Uma das principais aplicações é na construção de tabelas de _____, com plena utilização em compiladores e montadores.

Assinale a alternativa que apresenta as palavras que completam a sentença:

a) Armazenamento Associativo, tabelas e símbolos.
b) Vetor, índices, processos.
c) Armazenamento Associativo, índices e símbolos.
d) Vetor, tabelas, símbolos.
e) Armazenamento Associativo, tabelas e processos.

2. Dentro da estrutura de Armazenamento Associativo, a função de Mapas Associativos é realizar a associação entre elementos dentro da estrutura, realizando a associação entre uma chave e um valor recebidos, permitindo a recuperação rápida de um valor associado a uma chave.

Analise as sentenças a seguir:

I. Um mapeamento é definido por meio de uma relação entre seus pares.
II. Um mapeamento nem sempre é baseado em questões lógicas ou matemáticas.
III. Um mapa pode ser definido por fórmulas.
IV. Em uma função de mapeamento, não há retorno de valores.
V. Os Mapas Associativos permitem adicionarmos associações e removê-las se necessário.

Assinale a alternativa que apresenta as sentenças corretas quanto a Mapas Associativos:

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

3. Podemos concluir que a utilização de estruturas de Armazenamento Associativo está presente em todos os tipos de sistemas, seja em um simples sistema de cadastro de clientes de uma pequena loja, seja em um sistema de grande porte de uma empresa multinacional.

Assinale a alternativa que apresenta uma motivação para o uso de Armazenamento Associativo em estrutura de dados:

a) Poder desenvolver algoritmos básicos.

b) O fato de poder utilizar a linguagem C++ para criar algoritmos.

c) A criação de algoritmos com a utilização de vetores.

d) Poder desenvolver sistemas de associação para resolver problemas do mundo real.

e) Poder conhecer a estrutura na faculdade.

Seção 4.2

Mapas com Lista

Diálogo aberto

Caro aluno, você aprendeu na seção anterior a respeito das definições sobre Armazenamento Associativo, as motivações para sua utilização, os Mapas Associativos e casos de usos gerais.

Ao conhecer e compreender as estruturas de dados dinâmicas essenciais, bem como suas aplicações na solução de problemas em Mapas com Lista, você aprenderá:

- as definições e os exemplos de Mapas com Listas;

- a verificar a existência de uma chave em mapeamento com Listas;

- a adicionar e remover associações dadas suas chaves em mapeamento com Listas;

- a recuperar valores associados a chaves em mapeamento com Listas.

A utilização de Mapas com Listas permite implementar o armazenamento de informações de forma mais simples, facilitando a busca mais rápida de informações. Como exemplo da utilização dessa associação, temos os sites de cupons de desconto. Para a compra de uma promoção, você precisa ter seu cadastro realizado no site. Ao entrar no site e realizar a compra desse desconto, são associados ao número de sua compra seu nome, telefone e e-mail, assim como a compras dos demais clientes deste site, tendo o mesmo tipo de associação ao número de compra deles.

De posse do seu cupom, ao trocá-lo na respectiva loja, o comerciante, com seu número de compra, já tem acesso aos seus dados associados a ele, como nome, telefone e e-mail.

Desta forma, ao utilizar o mapeamento por meio de listas, podemos otimizar o sistema para a consulta e o armazenamento de informações associados a um índice.

Nesta seção, a sra. Fátima, nova empresária, está animada com as funcionalidades do sistema do restaurante que delegou a você. A principal função do sistema é controlar a venda dos pratos por número, dando baixa no estoque.

Com a solução da forma de criação do cardápio, a sra. Fátima solicitou agora que seja implementada no sistema a possibilidade de adicionar e remover pratos, assim como realizar suas consultas.

Como a estrutura de dados de mapeamento com listas pode resolver esse desafio? A adição e remoção de um prato nesse mapeamento podem ser realizadas de que forma? Como verificar se um prato já está cadastrado no sistema? E como o sistema vai recuperar as informações dos ingredientes e listá-las para o usuário? Como é possível realizar o controle dos pratos no sistema?

Será necessário analisar e levantar as formas possíveis de estrutura de dados que permitam criar uma solução para esse desafio.

Você precisará apresentar um relatório da possível implementação do trecho de código do sistema para esse desafio.

Prontos para solucionar esse desafio?

Não pode faltar

Prezado aluno, uma das formas mais simples de implementar um Mapeamento é guardar as associações pertencentes a ele dentro de uma Lista, podendo ser acessado por meio de um índice. Como exemplo de índice, podemos citar o RA (Registro Acadêmico) de um aluno, que é um índice para as informações pessoais relacionadas a ele e de uso da faculdade.

Definição e exemplos de Mapas com Listas

O uso de um índice é como uma referência associada a uma chave, utilizada para realizar a da otimização do desempenho e

permitindo uma localização mais rápida de um elemento quando solicitado, conforme mostra a Figura 4.4.

Figura 4.4 | Exemplo de um Mapa com Lista

Índice
1
2
3
4
5

→ Elemento

Chave	Informação
1	Azul
2	Amarelo
3	Vermelho
4	Verde
5	Preto

Fonte: elaborada pelo autor.

Na Figura 4.4, podemos observar a coluna de índice que se refere aos índices da Lista. A coluna de Elementos da Lista é formada por uma estrutura de mapa, em que é associada uma chave a uma informação. Desta forma, teremos um índice associado a uma chave contendo uma informação.

Exemplificando

Em banco de dados é utilizado um índice como uma estrutura auxiliar associada a uma tabela, com a função de acelerar o acesso às linhas de uma tabela, em que é criado um ponteiro para os dados armazenados. Desta forma, todas as linhas indexadas serão exibidas de forma mais rápida nas pesquisas realizadas.

Podemos utilizar como exemplo também a criação de um sumário de um livro. O sumário é criado com uma listagem de informações referente aos assuntos que serão tratados no livro, como podemos observar na Figura 4.5.

Figura 4.5 | Exemplo de sumário

A B C

Fonte: adaptada de Tenenbaum (1995).

Na Figura 4.5, é possível identificar os capítulos como índices da Lista (A), a descrição do assunto como a chave do nosso mapeamento (B) e a numeração da página como a informação mapeada pela chave (C). Assim, temos o índice para identificação dos capítulos de um livro e podemos verificar um assunto específico por meio desse índice e, depois, sua localização em páginas.

Assim como as demais Estruturas de Dados que estudamos até agora, a estrutura de Mapa com Lista também possui operações associadas a ela, como pesquisa de chave, adicionar ou remover associações e recuperar informações das chaves.

Assimile

Antes de adicionar ou remover uma associação no Mapa, o sistema deve verificar se a chave não pertence a alguma associação da Lista ou se está associada a um valor. Para realizar essa verificação, é necessário percorrer a Lista, pois o Mapa permite apenas associações com chaves distintas.

Verificação da Existência de uma Chave em Mapeamento com Listas

Para adicionar uma associação no Mapa, é necessário verificar se a chave da nova associação não pertence a alguma associação da Lista; para isso, precisamos verificar se a chave já existe na estrutura.

Para remover uma associação no mapeamento, também é necessário verificar se uma chave está associada a algum elemento ou recuperar o elemento associado, sendo necessário percorrer a Lista para verificar.

Antes de iniciarmos a implementação dos trechos de códigos dos nossos exemplos de verificação de existência de uma chave, assim como para adicionar ou remover uma associação e também para recuperar valores associados em uma chave em mapeamento com listas, é necessário declararmos as bibliotecas e a estrutura da Lista a seguir:

/* Manipula o fluxo de dados padrão do sistema, como entrada, saída e saída de erros; é uma evolução da biblioteca padrão <stdio.h> */

```
#include <iostream>
/* Responsável pela associação de objetos do tipo chave e elemento */
#include <map>

using namespace std;

/* Cria a estrutura da lista */
struct Materias{
    string codigo;
    string disciplina;
};
```

Pesquise mais

Para compreender melhor a estrutura de um Mapa com Listas, sugiro ler este artigo que apresenta informações sobre a estrutura de Mapa na forma de alocação contígua, que é a utilização de Listas. Consulte as páginas 5 a 9 neste link: http://calhau.dca.fee.unicamp.br/wiki/images/5/5b/Cap2.pdf. Acesso em: 25 jan. 2018.

RICARTE, I. L. M. Estruturas de dados (notas de aula) FEE-Unicamp. Texto completo disponível em: <https://www.passeidireto.com/arquivo/35042692/estruturas-de-dados---unicamp-notas-de-aula---prof-ivan-luiz-marques-ricarte>. Acesso em: 8 mar. 2018.

Desta forma, podemos implementar o trecho de código a seguir como um exemplo de verificação de existência de chave:

```
using namespace std;

/* Cria a estrutura da lista */
struct Materias{
    string codigo;
    string disciplina;
};
```

```
int main(){
    int chave;

    /* Cria o Mapa com Lista */
    map<int, Materias> MapaLista;

    Materias mat;

    /* Informa os dados para a associação */
    mat.codigo = "103";
    mat.disciplina = "ESTRUTURA DE DADOS I";
    MapaLista[1] = mat;
    mat.codigo = "203";
    mat.disciplina = "ESTRUTURA DE DADOS II";
    MapaLista[2] = mat;
    mat.codigo = "303";
    mat.disciplina = "ESTRUTURA DE DADOS III";
    MapaLista[3] = mat;

    printf("Informe a chave da disciplina: ");
    scanf("%s", &chave);

     /* Se achou a chave pesquisada, escreve Encontrou, caso contrário Não Encontrou */
    if(MapaLista.find(chave) == MapaLista.end())
        cout << "Nao Encontrado!\n" << endl;
    else
        cout << "Encontrado!\n"  << endl;

   return 0;
}
```

Adicionar/remover associações dadas suas chaves em mapeamento com listas

Como a estrutura de Mapa não pode permitir que duas associações com a mesma chave sejam inseridas e com a realização da verificação de existência de chave já realizada, podemos inserir uma nova chave em nossa estrutura de Mapa.

Utilizando como base o código de verificação de existência de chave, podemos implementá-lo com o trecho de código para adicionar uma nova associação de chave:

```
using namespace std;

/* Cria a estrutura da lista */
struct Materias{
    string codigo;
    string disciplina;
};

int main(){
    int chave;
    char disciplina[20], codigo[5];

    /* Cria o Mapa com Lista */
    map<int, Materias> MapaLista;

    Materias mat;

    mat.codigo = "103";
    mat.disciplina = "ESTRUTURA DE DADOS I";
    MapaLista[1] = mat;
    mat.codigo = "203";
    mat.disciplina = "ESTRUTURA DE DADOS II";
    MapaLista[2] = mat;
    mat.codigo = "303";
    mat.disciplina = "ESTRUTURA DE DADOS III";
```

```
        MapaLista[3] = mat;

        printf("Adicionar nova disciplina\n");
        printf("Informe a chave para armazenar: ");
        scanf("%d", &chave);

        /* Verifica se a chave não existe */
        if(MapaLista.find(chave) == MapaLista.end()){
            /* Informa os dados para a associação caso a chave não
exista*/

            printf("Informe o codigo da disciplina: ");
            scanf("%s", &codigo);

            printf("Informe o nome da disciplina: ");
            scanf("%s", &disciplina);

            /* Insere no mapa */
            mat.codigo = codigo;
            mat.disciplina = disciplina;
            MapaLista[chave] = mat;

            /* Imprime na tela os dados da disciplina inserido */
            cout << "\nDisciplina Inserida com sucesso"  << endl;
                cout << "\nCodigo: " + MapaLista[chave].codigo + "\
nDisciplina: " + MapaLista[chave].disciplina + "\n"  << endl;
        } else {

            cout << "Chave ja existente!\n" << endl;
}

       return 0;
    }
```

Para remover uma associação, comparamos a chave informada com as demais chaves das associações da Lista. Se acaso a chave existir, será removida do mapeamento, deixando essa posição disponível para uma eventual adição com essa chave.

Podemos utilizar o trecho de código a seguir para implementar a remoção da associação com base na chave informada pelo usuário:

```
printf("Informe a chave para remocao: ");
scanf("%d", &chave);

/* Verifica se a chave existe */
if(MapaLista.find(chave) == MapaLista.end()){
    cout << "Chave nao existente!\n" << endl;
} else {
   /*Remove a associação da chave informada pelo usuário */
    MapaLista.erase(MapaLista.find(chave));
}
```

Reflita

No exemplo de trecho de código apresentado anteriormente para adicionar e remover uma associação de Mapa com Lista, utilizamos o código simples diretamente na função principal ***Main( ),*** sem uma função para executar a adição ou remoção. É possível utilizarmos uma função específica passando o Mapa como parâmetro?

Recuperar valores associados a chaves em mapeamento com listas

A recuperação de valores associados a chaves em um mapeamento com listas pode ser implementada de forma bem simples, utilizando a mesma lógica de verificação de existência de uma chave, alterando apenas o retorno da mensagem, como apresentado no trecho de código a seguir:

```
printf("Informe a chave para pesquisar: ");
scanf("%d", &chave);
```

```
        /* Verifica se a chave existe */
        if(MapaLista.find(chave) == MapaLista.end()){
            cout << "Disciplina nao encontrada!\n" << endl;

        } else {
            /* Caso exista, a disciplina retorna as informações na lista associada à chave */
        cout << "Disciplina Encontrada!\nCodigo: " + MapaLista[chave].codigo + "\nDisciplina: " + MapaLista[chave].disciplina + "\n" << endl;
        }
```

Outra forma de utilizarmos essa pesquisa de elementos em um mapeamento é realizar a pesquisa por meio do código da disciplina, no caso do nosso exemplo, ou seja, por um dos elementos da Lista, que nos apresentaria como resultado final o mesmo do trecho de código anterior:

```
        printf("Informe o codigo da disciplina: ");
        scanf("%s", &coddisc);

        /* Realiza um loop para verificar se o nome foi encontrado com um contador i enquanto for inferior ou igual ao tamanho do Mapa */
        for (i = 1; i <= MapaLista.size(); i++) {
            printf("Posicao %d: ", i);

            /* Se o nome pesquisado existir, escreve a posição e os dados da chave */
            if(MapaLista[i].codigo == coddisc)
                /* Caso exista, a disciplina retorna as informações na lista associada à chave */
            cout << "Disciplina Encontrada!\nCodigo: " + MapaLista[i].codigo + "\n Disciplina: " + MapaLista[i].disciplina + "\n" << endl;
            else

                cout << "Disciplina nao encontrada!\n" << endl;
}
        }
```

Neste caso, precisaríamos criar uma variável para ser o contador; neste exemplo, a variável é do tipo ***char coddisc[5]***, para receber o código da disciplina a ser pesquisada. Utilizamos a função ***MapaLista.size()*** para identificar o tamanho do mapa.

A utilização de Mapa com Lista permite um acesso mais simples e rápido aos dados associados.

Sem medo de errar

Retornando ao nosso desafio desta seção, a sra. Fátima, nova empresária, está animada com as funcionalidades do sistema do restaurante que delegou a você, tendo como principais funções do sistema o controle de venda dos pratos por número e a baixa no estoque.

Com a solução da forma de criação do cardápio, a sra. Fátima solicitou agora que seja implementada no sistema a possibilidade de adicionar e remover pratos no sistema, assim como realizar suas consultas.

Como a estrutura de dados de mapeamento com listas pode resolver esse desafio? A adição e remoção de um prato nesse mapeamento podem ser realizadas de que forma? Como verificar se um prato já está cadastrado no sistema? Como o sistema vai recuperar as informações dos ingredientes e listá-las para o usuário? Como é possível realizar o controle dos pratos no sistema?

Será necessário analisar e levantar as formas possíveis de estrutura de dados que permitam criar uma solução para esse desafio.

Você precisará apresentar um relatório da possível implementação do trecho de código do sistema para esse desafio.

Como solução para nosso desafio, podemos apresentar a seguinte implementação deste sistema:

```
#include <iostream>
#include <map>

using namespace std;
/* Cria a estrutura da lista */
```

```
struct Restaurante{
    string descprato;
    string ingredprato;
};

int main(){
     int i = 0, chave, opcao;
     char descricao[15], ingr[50];

     /* Cria o Mapa com Lista */
     map<int, Restaurante> MapaPratos;

    Restaurante pratos;

     do {
          system("cls");
          printf("Selecione uma opcao:\n");
          printf("1 - Adicionar um Prato\n");
          printf("2 - Remover um Prato\n");
          printf("3 - Listar todos os Pratos\n");
          printf("4 - Sair\n");

          scanf("%d", &opcao);
          switch (opcao){
               case 1:
                    system("cls");
                    printf("Adicionar nova disciplina\n");
                    printf("Informe o nome do prato: ");
                    scanf("%s", &descricao);
                    pratos.descprato = descricao;
```

```
                printf("\nInforme os ingredientes do prato: ");
                /* A função gets recebe o texto todo da variável ingr,
incluindo espaço */
                gets(ingr);

                pratos.ingredprato = ingr;
                MapaPratos[(MapaPratos.size()+1)] = pratos;

                cout << "\nPrato adicionado com sucesso!" << endl;
                cout << "\nPrato: " + MapaPratos[MapaPratos.size()].
descprato + "\nIngredientes: " + MapaPratos[MapaPratos.size()].
ingredprato + "\n" << endl;
                system("pause");
                opcao = 0;
                break;

            case 2:
                system("cls");
                printf("Informe o numero do prato para remover: ");
                scanf("%d", &chave);

                /* Verifica se o número do prato existe */
                if(MapaPratos.find(chave) == MapaPratos.end()){
                    cout << "Prato nao existente!\n" << endl;
                } else {
                    MapaPratos.erase(MapaPratos.find(chave));
                }

                system("pause");
                opcao = 0;
                break;
```

```
            case 3:
                system("cls");
                printf("*** Listagem dos pratos! ***\n");

                for (i = 1; i <= MapaPratos.size(); i++) {
                    printf("Prato %d: ", i);
                  /* Se o prato pesquisado existir, escreve a posição
e os dados da chave */
                    if(MapaPratos.find(i) == MapaPratos.end())
                        cout << "Prato nao encontrada!\n" << endl;
                    else
                        /* Caso exista o prato, retorna as informações
na lista associada à chave */
                        cout << "\nNome: " + MapaPratos[i].descprato
+ "\nIngrediente: " + MapaPratos[i].ingredprato + "\n"  << endl;
                }

                system("pause");
                opcao = 0;
                break;

            default:
                break;
        }
    }while (opcao != 4);
    return 0;
}
```

Avançando na prática

Resultado de votação

Descrição da situação-problema

Uma empresa que realiza prestação de serviços para condomínios, ao saber do seu conhecimento em desenvolvimento de sistemas

dinâmicos, entrou em contato com você para o desenvolvimento e a implementação de um sistema que ela pretende oferecer aos seus clientes que possuem condomínios.

Esses condomínios realizam eleições para síndicos ou para responsável pela sua gestão. No entanto, muitos proprietários e moradores não comparecem às reuniões por diversos motivos. Com o intuito de realizar um sistema de votação mais abrangente, essa empresa solicitou a você a criação desse sistema, que consiste em uma tela em que os moradores ou proprietários acessam uma lista com os candidatos a síndicos e seus respectivos números para votação. Os números estão associados ao nome do candidato e à quantidade de votos conquistados.

Utilizando o Mapa com Listas, você precisa implementar esse sistema e apresentar à empresa responsável.

Resolução da situação-problema

Para a resolução desse desafio, você pode implementar um exemplo de código como:

```
#include <iostream>
#include <map>

using namespace std;

/* Cria a estrutura da lista */
struct Candidatos{
    string nomecand;
    int totalvotos;
};

int main(){
    int i = 0, chave, opcao;
    char nome[15];

    /* Cria o Mapa com Lista */
    map<int, Candidatos> MapaVotos;
```

```
        Candidatos votos;

        do {
            system("cls");
            printf("Selecione uma opcao:\n");
            printf("1 - Adicionar um Candidato\n");
            printf("2 - Votar\n");
            printf("3 - Resultado parcial\n");
            printf("4 - Sair\n");

            scanf("%d", &opcao);

            switch (opcao){
                case 1:
                    system("cls");
                    printf("Adicionar novo candidato\n");
                    printf("Informe o nome do candidato: ");
                    scanf("%s", &nome);

                    votos.nomecand = nome;
                    votos.totalvotos = 0;
                    MapaVotos[(MapaVotos.size()+1)] = votos;

                     cout << "\nCandidato adicionado com sucesso!" <<
endl;
                     cout << "\nNome: " + MapaVotos[MapaVotos.size()].
nomecand + "\nTotal de votos: " << MapaVotos[MapaVotos.size()].
totalvotos << "\n" << endl;

                    system("pause");
                    opcao = 0;
                    break;
```

```
                case 2:
                        system("cls");
                        printf("Vote pelo numero:\n");

                      for (i = 1; i <= MapaVotos.size(); i++) {
                          printf("%d - ", i);
                          cout << MapaVotos[i].nomecand << endl;
                    }

                     scanf("%d", &chave);

                     /* Verifica se o número do candidato existe */
                     if(MapaVotos.find(chave) == MapaVotos.end()){
                         cout << "Candidato nao existe!\n" << endl;
                     } else {
                        MapaVotos[chave].totalvotos = MapaVotos[chave].
totalvotos + 1;
                         cout << "Voto realizado com sucesso!\n" << endl;
                     }

                     system("pause");
                     opcao = 0;
                     break;

                case 3:
                 system("cls");
                    printf("*** Resultado Parcial! ***\n");

                 for (i = 1; i <= MapaVotos.size(); i++) {
                     printf("Candidato numero %d: ", i);
```

```
                /* Se o candidato pesquisado existir, escreve a posição e os dados da chave */
                if(MapaVotos.find(i) == MapaVotos.end())
                    cout << "Candidato nao encontrado!\n" << endl;
                else
                    /* Caso exista o candidato, retorna seu nome e o total de votos da lista associada à chave */
                    cout << "\nNome: " + MapaVotos[i].nomecand + "\nTotal de Votos: " << MapaVotos[i].totalvotos << "\n"  << endl;
                }

                system("pause");
            opcao = 0;
            break;

            default:
                break;
            }
        }while (opcao != 4);

        return 0;
    }
```

Faça valer a pena

1. Os _____ são estruturas de dados que implementam o tipo de situação de mapeamento em _____. Uma das formas mais simples de implementar um Mapeamento é guardar as associações pertencentes a ele dentro de uma _____, podendo ser acessado por meio de um _____.

Assinale a alternativa que apresenta as palavras corretas, respectivamente, para preenchimento das lacunas:

a) Banco de dados, sequência, tabela e link.
b) Mapas, associação, Lista e link.
c) Mapas, sequência, tabela e índice.
d) Mapas, associação, Lista e índice.
e) Bancos de dados, associação, Lista e link.

2. Para remover uma associação, comparamos a chave informada com as demais chaves das associações da Lista. Se acaso a chave existir, será removida do mapeamento, deixando essa posição disponível para uma eventual adição com essa chave.

Assinale a alternativa que apresenta a linha de comando que permite a exclusão da associação de um Mapa com Lista:

a) MapaLista.delete(MapaLista.find(chave)).
b) MapaLista.erase(MapaLista.find(chave)).
c) MapaLista.remove(MapaLista.find(chave)).
d) MapaLista.out(MapaLista.find(chave)).
e) MapaLista.exit(MapaLista.find(chave)).

3. Para adicionar uma associação ao Mapa, é necessário verificar se a chave da nova associação não pertence a alguma associação da Lista. Para isso, precisamos verificar se a chave já existe na estrutura, assim como para remover uma associação, é necessário verificar se uma chave está associada a algum elemento; para recuperar o elemento associado, é necessário percorrer a Lista para verificar sua existência.

Assinale a alternativa que apresenta restrição da utilização de Mapas com Lista:

a) O Mapa permite utilizar apenas o índice da Lista para verificar a existência de uma chave.
b) O Mapa permite apenas associações com chaves distintas.
c) O Mapa permite no máximo duas associações com a mesma chave.
d) O Mapa permite diversas associações com a mesma chave.
e) O Mapa permite apenas Lista Circulares em sua estrutura.

Seção 4.3

Mapas com Espalhamento

Diálogo aberto

Caro aluno, na seção anterior você conheceu e aprendeu Mapas com Listas, sua definição e exemplos, a existência de uma chave em mapeamentos com listas, as operações para adicionar ou remover uma associação com base na chave de mapeamento e a recuperação de valores associados a chaves de mapeamento em lista.

Nesta seção, sobre os Mapas com Espalhamento, você:

- compreenderá sua definição;
- conhecerá exemplos de aplicação dessa estrutura;
- verificará a existência de uma chave em associação com os Mapas com Espalhamento.

Você também conhecerá e aprenderá como:

- adicionar ou remover associações com base em Chave nos Mapas com Espalhamento;
- recuperar valores associados a uma chave de Mapas com Espalhamento.

A utilização de Mapa com Espalhamento permite uma associação entre chaves e valores mais rápida e, assim, quando empregado em sistemas com muitas informações, possibilita a rápida identificação dos elementos associados. Imagine a utilização de um sistema de cadastro de CPF (Cadastro de Pessoa Física) em âmbito nacional, como um *e-commerce*, por exemplo. O acesso simultâneo ao sistema é grande. Assim o acesso de associações entre o CPF e o nome do cliente precisa ser rápido, tanto no cadastro como na recuperação de informações associadas ao CPF.

Você estudou na unidade anterior as Tabelas de Espalhamento e sua funcionalidade e, nesta seção, poderá compreender a utilização

de mapeamento com as tabelas de espalhamento, melhorando, assim, a performance da estrutura.

Retornando ao nosso desafio desta unidade, você está progredindo muito com o desenvolvimento do sistema de restaurante contratado pela Sra. Fátima e, após a análise e o levantamento de como utilizar as estruturas de dados para solucionar a questão de adicionar ou remover os pratos por número, chegou a sua vez de melhorar o sistema, verificando o que mais será preciso otimizar para suas buscas.

A estrutura de dados de mapeamento de espalhamento pode auxiliar na melhora e otimização do sistema. Desta forma, o sistema será mais ágil ao cadastrar pratos e ao recuperar essas informações quando possuir muitos dados cadastrados. Pesquise e realize um levantamento de como o mapa de espalhamento pode resolver a adição e a remoção de pratos, assim como a pesquisa de pratos controlados por números e associados a todos os ingredientes neles contidos.

Para concluir o desafio, é preciso entregar um documento de projeto indicando como o espalhamento pode auxiliar no uso e na implementação do código utilizando a associação com espalhamento.

Pronto para concluir mais esse desafio? Vamos lá!

Não pode faltar

Prezado aluno, a implementação de estrutura de dados do tipo Mapa, utilizando Listas, pode ser menos eficiente quando o sistema possui muitas informações, pois, em operações em que precisam ser percorridas todas as associações e, conforme o número de associações cresce, o tempo para percorrer a estrutura torna-se mais demorado, interferindo no desempenho do sistema.

Definição e exemplos de Mapas com Espalhamento

Para Tenenbaum (1995), da mesma forma que podemos utilizar a técnica de Espalhamento para melhorar o desempenho da estrutura de dados para acesso rápido às informações, podemos utilizar

a técnica de Espalhamento associada aos Mapas para suprir essa necessidade de melhora nos acessos.

Conforme Goodrich (2013), assim como em Mapas com Listas, em Mapas com Espalhamento as chaves das associações também não podem se repetir.

Assimile

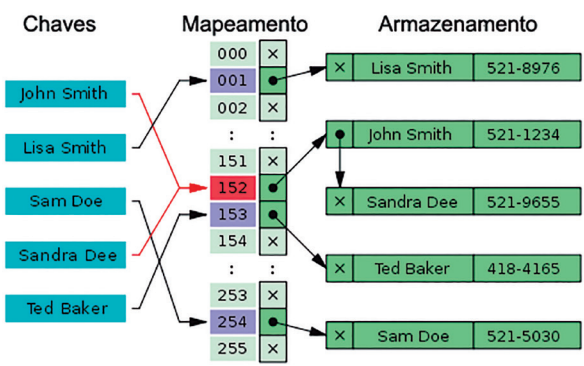
Você estudou que, para utilizar a técnica de Espalhamento, precisamos definir a Função de Espalhamento e a Tabela de Espalhamento, e o Código de Espalhamento não é gerado com base na estrutura de dados, mas, sim, com base no próprio elemento, no caso, as chaves.

O Mapa com Espalhamento tem como características:

- os elementos não são ordenados;
- rápida busca/inserção de dados;
- permite inserir valores e chaves nulas.

O Mapa com Espalhamento permite gerenciar uma sequência de elementos como uma Tabela de Espalhamento, com cada entrada de tabela armazenando uma lista de nós vinculados e cada nó armazenando um elemento. Um elemento consiste em uma chave, para poder ordenar a sequência e um valor mapeado, conforme indica a Figura 4.6.

Figura 4.6 | Exemplo de Mapa com Espalhamento



Fonte: adaptada de: <http://www.codenuclear.com/difference-between-hashmap-hashtable/>. Acesso em: 6 mar. 2018.

Um Mapa com Espalhamento é uma estrutura de Armazenamento Associativo. Seus elementos são agrupados com base na aplicação da Função de Espalhamento na chave dos elementos, gerando, assim, a associação entre elemento e chave.

A utilização do Mapa com Espalhamento é muito comum quando se trabalha com valores "nomeados", ou seja, não importa a posição do elemento, mas o valor da sua chave.

Exemplificando

O Mapa com Espalhamento é utilizado com muita frequência no sistema de parametrização de métodos, ou seja, um sistema que tenha um método que pode receber um número diversificado de parâmetros com grande quantidade de nomes distintos.

Trata-se de um sistema que possui uma tela de configuração de parâmetros, em que o usuário pode configurar o parâmetro para enviar um e-mail ou não, outro parâmetro para liberar o acesso externo de usuário ou não, ou um parâmetro para o sistema pegar a hora automaticamente ou ser informado pelo usuário.

Para criarmos a estrutura do nosso Mapa de Espalhamento, precisamos realizar o trecho de código a seguir, como neste exemplo:

```
/* Criamos a estrutura de associação da chave de Espalhamento */
typedef struct HashmapNo {
    unsigned int hash_index;
    int valor;
} HashmapNo;

/* Definimos a estrutura da Tabela de Espalhamento */
typedef struct HashMapa {
    int cont_elemento;
    int map_size;
    HashmapNo ** node_list;
} HashMapa;
```

Após criada e implementada a estrutura do Mapa de Espalhamento, é necessário inicializá-la, como no exemplo do trecho de código a seguir:

```
HashMapa* hashmap_new(int size) {
    int i;
  HashMapa *hashmapa = (HashMapa*)malloc(sizeof(HashMapa));
    hashmapa -> node_list = malloc(size * sizeof(HashmapNo*));
    hashmapa -> cont_elemento = 0;
    hashmapa -> map_size = size;

    for (i = 0; i < size; i++) {
        hashmapa -> node_list[i] = malloc(sizeof(HashmapNo));
        hashmapa -> node_list[i] = NULL;
    }
    return hashmapa;
}
```

Pesquise mais

A classe <hash_map> ajuda a gerenciar uma sequência de elementos como uma Tabela de Espalhamento e possui compatibilidade com versões anteriores, como a classe <hash_map.h>. Para conhecer mais sobre essa classe, acesse este link:

CODECogs. Hash Maps. Disponível em: <http://www.codecogs.com/library/computing/stl/containers/associative/hash-map.php>. Acesso em: 6 mar. 2018.

Verificação da existência de uma Chave em Mapas com Espalhamento

Segundo Goodrich (2013), como estamos utilizando a técnica de Espalhamento para verificar a existência de uma chave no Mapa, precisamos calcular o índice da Tabela e procurá-lo na Lista correspondente, como podemos observar na implementação do código a seguir:

```
    /* Calculamos o valor da chave */
    unsigned int hashmapa_hash_func(HashMapa *hashmapa, unsigned int key) {
        int hash = key;

        /* Calcula o valor da chave para evitar a colisão nas chaves */
        hash = (hash >> 3) * 2654435761;
        hash = hash % hashmapa -> map_size;
        return hash;
    }

    /* Função para verificar a existência
    HashmapNo* hashmapa_verifica(HashMapa *hashmapa, unsigned int key) {
        /* Criamos a variável do tipo inteiro para receber o resultado da função hashmapa e informamos que é unsigned, ou seja, será sempre positivo o valor */
        unsigned int hash = hashmapa_hash_func(hashmapa, key);

        /* É criada a variável hashmap_node para receber o valor da chave */
        HashmapNo *hashmap_node = hashmapa -> node_list[hash] ;

        /* Caso o valor recebido em hashmap_node seja nulo, então não haverá chave associada, caso contrário, informará que a chave foi encontrada */
        if (hashmap_node == NULL)
            return "Chave não encontrada!";
        else
            return "Chave encontrada!";
    }
```

Adicionar/remover associações dadas suas Chaves em Mapas com Espalhamento

Para Goodrich (2013), ao adicionar uma nova associação, pode ser que a chave já exista no Mapa. Neste caso, vamos retirar a associação antiga antes de colocar a nova. Isso deve ser feito porque o Mapa não permite chaves repetidas. O trecho de implementação de código a seguir apresenta um modelo de inserção:

```
/* Função para inserir uma associação */
void hashmapa_insere(HashMapa *hashmapa, unsigned int key, int valor) {
    unsigned int hash = hashmapa_hash_func(hashmapa, key);
    HashmapNo *hashmap_node = hashmapa -> node_list[hash];

    if (hashmap_node != NULL && hashmapa -> node_list[hash] -> hash_index == key)  {
        hashmapa_ remove (hashmapa, key);

    hashmap_node = hashmapa_new_node(hash, valor);
    hashmapa -> cont_elemento++;
    }
}
```

Já a remoção de uma associação é um procedimento simples: apenas calculamos o índice e procuramos a chave na tabela correspondente e, ao encontrarmos a chave, removemos a associação. Como exemplo de implementação da remoção de uma associação, temos o seguinte trecho de código:

```
/* Função para remover uma associação */
void hashmapa_remove(HashMapa *hashmapa, int key) {
    int hash = hashmapa_hash_func(hashmapa, key);
    HashmapNo *hashmap_node = hashmapa -> node_list[hash];
    hashmap_node -> values = NULL;
    hashmap_node -> hash_index = 0;
    hashmap_node -> values_size = 0;
}
```

Recuperar valores associados a chaves em Mapas com Espalhamento

Conforme Goodrich (2013), o principal objetivo do Mapa com Espalhamento é oferecer uma forma rápida de acessar o valor de uma chave desejada e, assim, ter um desempenho da estrutura maior que o das demais estruturas.

Então, precisamos procurar a associação da chave na tabela adequada e devolver o valor correspondente. Se a chave não existir, retornaremos ao usuário a mensagem de que a chave não foi encontrada.

Podemos utilizar o trecho de código a seguir para recuperar as informações associadas à chave desejada, como realizar uma pesquisa no sistema para saber quais os dados de endereço de uma pessoa, consultando-os pela chave, que pode ser seu nome ou seu CPF (Cadastro de Pessoa Física):

```
/* Função para recuperar os valores da tabela associada */
HashmapNo* hashmapa_verifica(HashMapa *hashmapa, unsigned int key) {
    unsigned int hash = hashmapa_hash_func(hashmapa, key);
    HashmapNo *hashmap_node = hashmapa -> node_list[hash];

    if (hashmap_node == NULL)
        return "Chave não encontrada!";
    else
        return hashmapa -> node_list[hash];
}
```

Reflita

Utilizando a técnica de Espalhamento, podemos obter um consumo de tempo médio constante para todas as operações. De que forma a técnica de Espalhamento associada ao mapeamento pode ser mais eficiente do que a associação do mapeamento com Listas?

Porém, ainda precisaríamos tratar o problema da sobrecarga das Listas da Tabela de Espalhamento. Podemos transformar a Tabela de Espalhamento em uma tabela dinâmica, ou seja, em uma tabela que aumenta e diminui de tamanho conforme a necessidade do Mapa, utilizando uma função para expandi-la. Podemos utilizar a implementação de um trecho de código de expansão como o mostrado a seguir:

```
/* Função para transformar a Tabela de Espalhamento em tabela dinâmica */
void hashmapa_expand(HashMapa *hashmapa) {
    int i;
    int hash_index;
    int old_size = hashmapa -> map_size;
    HashmapNo *hashmap_node;

   /* Cria a alocação de memória, duplicando o espaço necessário */
    hashmapa -> map_size = old_size * 2;
     HashmapNo **new_node_list = malloc(hashmapa -> map_size * sizeof(HashmapNo*));

      /* Utiliza-se um for para inicializar o novo espaçamento alocado */
    for (i = 0; i < hashmapa -> map_size; i++) {
        new_node_list[i] = malloc(sizeof(HashmapNo));
        new_node_list[i] = NULL;
    }

   /* Utiliza-se o for para transferir os valores da alocação existente para a nova alocação enquanto houver valores nela alocados */
    for (i = 0; i < old_size; i++) {
        hashmap_node = hashmapa -> node_list[i];
```

```
            if(hashmap_node != NULL) {
                    hash_index = hashmapa_hash_func(hashmapa,
hashmap_node -> hash_index);
                hashmap_node -> hash_index = hash_index;
                new_node_list[hash_index] = hashmap_node;
            }
            /* Libera a alocação anterior */
            free(hashmapa_node);
        }
        /* Mapeamento recebe a nova alocação dinâmica */
        hashmapa -> node_list = new_node_list;
    }
```

Os Armazenamentos Associativos com Espalhamento são otimizados para as operações de pesquisa, inserção e remoção e são muito eficientes quando usados com uma Função de Espalhamento bem projetada, executando-as em um tempo que é, em média, constante e não depende do número de elementos. Uma Função de Espalhamento bem projetada realiza uma distribuição uniforme de valores e minimiza o número de colisões, e, no pior caso, com a pior Função de Espalhamento possível, o número de operações será proporcional ao número de elementos na sequência de elementos.

Chegamos ao final dos nossos estudos sobre Algoritmos e Estrutura de Dados. Você pôde aprender como as estruturas funcionam e como são importantes no dia a dia da programação e no desenvolvimento de sistemas.

É muito importante saber que esta é apenas uma parte de Estrutura de Dados e, quanto mais você buscar e adquirir conhecimento nas estruturas avançadas, melhor será seu desenvolvimento técnico e profissional como solucionador de problemas do mundo real.

Sem medo de errar

Retomando o nosso último desafio desta unidade, você progrediu muito com o desenvolvimento do sistema de restaurante contratado pela Sra. Fátima e, após a análise e o levantamento de como utilizar as estruturas de dados para solucionar a questão de

adicionar ou remover os pratos por número, chegou a sua vez de melhorar o sistema, verificando o que mais será preciso otimizar nas buscas do sistema.

As estruturas de dados de mapeamento de espalhamento podem auxiliar na melhora e otimização do sistema. Pesquise e realize um levantamento de como o mapa de espalhamento pode resolver a adição e a remoção de pratos, assim como a pesquisa de pratos controlados por números e associados a todos os ingredientes neles contidos.

Para encerrar o desafio, é preciso entregar um documento de projeto indicando como o espalhamento pode auxiliar o uso e a implementação do código utilizando a associação com espalhamento.

Para concluir seu último desafio, a implementação do trecho de código a seguir, utilizando a associação com espalhamento do sistema do Restaurante da Sra. Fátima, pode ser realizada por:

```
#include <hash_map>
#include <iostream>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct NumPratos {
    unsigned int hash_index;
    int valor;
} NumPratos;

typedef struct Pratos {
    int cont_elemento;
    string descricao;
    int map_size;
    NumPratos ** num_list;
} Pratos;
```

```
NumPratos* novo_num_prato(unsigned int hash_index,int values_size) {
  NumPratos *num_prato = malloc(sizeof(NumPratos));
  num_prato -> hash_index = hash_index;
  num_prato -> values_size = values_size;
  num_prato -> values = malloc(sizeof(values_size * sizeof(int)));

  return num_prato;
}

Pratos* novo_prato(int size) {
  int i;
  Pratos *pratos = (Pratos*)malloc(sizeof(Pratos));
  pratos -> num_list = malloc(size * sizeof(NumPratos*));
  pratos -> cont_elemento = 0;
  pratos -> descricao = "";
  pratos -> map_size = size;

  for(i = 0; i < size; i++) {
    pratos -> num_list[i] = malloc(sizeof(NumPratos));
    pratos -> num_list[i] = NULL;
  }
  return pratos;
}

unsigned int prato_hash_func(Pratos *pratos, unsigned int key) {
  int hash = key;

  hash = (hash >> 3) * 2654435761;
  hash = hash % pratos -> map_size;
  return hash;
}
```

```
void remove_prato(Pratos *pratos, int key) {
  int hash = prato_hash_func(pratos, key);
  NumPratos *num_pratos = pratos -> num_list[hash];
  num_pratos -> values = NULL;
  num_pratos -> hash_index = 0;
  num_pratos -> values_size = 0;
}

NumPratos* pesquisa_prato(Pratos *pratos, unsigned int key) {
  unsigned int hash = prato_hash_func(pratos, key);
  NumPratos *num_pratos = pratos -> num_list[hash];

  if(num_pratos == NULL) {
    return NULL;
  } else {
    return pratos -> num_pratos[hash];
  }
}

void insere_prato(Pratos *pratos, unsigned int key, int valor) {
  unsigned int hash = prato_hash_func(pratos, key);
  NumPratos *num_pratos = pratos -> num_list[hash];

   if(num_pratos != NULL && pratos -> num_list[hash] ->hash_
index == key) {
    return;
  } else {
    num_pratos = novo_prato(hash, valor);
    pratos -> cont_elemento++;
  }
}
```

Avançando na prática

Cadastro de veículos

Descrição da situação-problema

Você se formou na faculdade e, devido ao seu grande destaque com programação e estrutura de dados, foi contratado como desenvolvedor de sistemas d e uma empresa de software. Ao entrar na empresa, seu gestor lhe imcubiu de cuidar da conta de um novo cliente.

Esse novo cliente, o Sr. Raul, possui um despachante de documentos de veículos e gostaria de ter um sistema para cadastrar os veículos sob sua responsabilidade. Ele deseja que, ao ser digitada a placa do veículo, o sistema retorne todos os dados cadastrados, assim como os dados do cliente e os do proprietário do veículo. Seu desafio é realizar um levantamento do que é necessário implementar no sistema para suprir a necessidade do sr. Raul, mantendo um sistema com ótimo desempenho. Ao concluir, você deve entregar um relatório com as informações levantadas do sistema.

Resolução da situação-problema

Para resolver esse novo desafio e auxiliar o sr. Raul com o desenvolvimento de um sistema que suprirá sua necessidade de cadastrar seus clientes em seu despachante de documentos, por meio das placas dos veículos, utilizaremos os Mapas com Espalhamento.

Por meio do Mapa de Espalhamento, é possível criar as associações entre a placa dos veículos, seus dados e os dos seus clientes.

Assim, o levantamento do sistema para desenvolvimento deve ser realizado com base em:

- criar a estrutura base do Mapa com Espalhamento;
- criar a função para inicializar a estrutura;
- definir a placa do carro como a chave da estrutura;
- implementar a função para gerar o código de espalhamento;

- implementar a função para criar uma nova associação;

- desenvolver a função para remover uma associação;

- criar a função de consulta dos dados;

- implementar a função principal para o Sr. Raul realizar os lançamentos das informações necessárias e específicas que deseja.

Faça valer a pena

1. A estrutura de dados de _____ permite gerenciar uma sequência de elementos como uma _____, e com cada entrada da tabela armazenando uma lista de _____ vinculados, com cada nó armazenando um elemento. Um elemento consiste em uma _____, para poder ordenar a sequência e um valor mapeado.

Assinale a alternativa que apresenta as sentenças que completam as lacunas:

a) Mapa com Lista, Tabela de Espalhamento, dados e informação.
b) Tabela de Espalhamento, Pilha, nós, informação.
c) Mapa com Lista, Lista, dados, chave.
d) Mapa com Espalhamento, Fila, elementos, informação.
e) Mapa com Espalhamento, Tabela de Espalhamento, nós, chave.

2. Da mesma forma que podemos utilizar a técnica de Espalhamento para melhorar o desempenho da estrutura de dados para acesso rápido às informações, podemos utilizar a técnica de Espalhamento associada aos Mapas para suprir essa necessidade de melhora nos acessos.

Com referência às características do Mapa com Espalhamento, analise as sentenças a seguir:

I. Os elementos não são ordenados.
II. As chaves das associações podem se repetir.
III. Pode mapear valores para chaves.
IV. É rápido na busca/inserção de dados.
V. Permite inserir valores e chaves nulas.

Assinale a alternativa que apresenta as sentenças verdadeiras.

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

3. Um Mapa com Espalhamento é uma estrutura de Armazenamento Associativo e seus elementos são agrupados em associações com base no valor de uma Função de Espalhamento aplicada nos valores-chave dos elementos.

A utilização do Mapa com Espalhamento é muito comum quando se trabalha com valores "nomeados", ou seja, não importando a posição do elemento, mas, sim, o valor da sua chave.

Assinale a alternativa que pode apresentar um exemplo de uso de Mapa com Espalhamento:

a) Tela de cadastro de cliente.

b) Tela de configuração de sistema.

c) Tela de visualização de senha.

d) Sistema de empilhamento de estoque.

e) Tela de histórico de vendas.

Referências

CODENuclear. **Difference between HashMap and Hashtable**. Disponível em: <http://www.codenuclear.com/difference-between-hashmap-hashtable/>. Acesso em: 6 mar. 2018.

GOODRICH, M. T.; TAMASSIA, R. **Estruturas de dados & algoritmos em Java**. 5. ed. São Paulo: Bookman, 2013.

PEREIRA, S. L. **Estrutura de dados fundamentais**: conceitos e aplicações. 12. ed. São Paulo: Érica, 2008.

TENENBAUM, A. M. **Estruturas de dados usando C**. São Paulo: Makron Books, 1995.

ISBN 978-85-522-0660-6