ESTRUTURA DE DADOS

AUTOR
RICARDO BALIEIRO

ESTRUTURA DE DADOS

AUTOR
RICARDO BALIEIRO

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2015

Conselho editorial REGIANE BURGER; ROBERTO PAES; GLADIS LINHARES; KAREN BORTOLOTI; HELCIMARA AFONSO DE SOUZA

Autor do original RICARDO LUIS BALIEIRO

Projeto editorial ROBERTO PAES

Coordenação de produção GLADIS LINHARES

Coordenação de produção EaD KAREN FERNANDA BORTOLOTI

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística ROSELI CANTALOGO COUTO

Imagem de capa ARTUR MARCINIEC | DREAMSTIME.COM

Dados Internacionais de Catalogação na Publicação (CIP)

B186E BALIEIRO, RICARDO

Estrutura de dados / Ricardo Balieiro.

Rio de Janeiro : SESES, 2015.

176 p. : il.

ISBN: 978-85-60923-34-2

1. Lista sequencial. 2. Lista encadeada. 3. Pilha. 4. Fila. I. SESES. II. Estácio.

CDD 005.1

Diretoria de Ensino — Fábrica de Conhecimento

Rua do Bispo, 83, bloco F, Campus João Uchôa

Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio

Prezados(as) alunos(as),

No projeto de um sistema, vários fatores influenciam na sua qualidade final. Entre estes fatores, estão os mais importantes que são os profissionais envolvidos no seu desenvolvimento do sistema. O profissional necessita ter sólidos conhecimentos, que vão além dos recursos da linguagem de programação, utilizados durante todo o projeto, como também das estruturas de dados, que são fundamentais em qualquer projeto que envolva manipulação de dados. Desta forma é importante que tenhamos uma visão detalhada dos mecanismos envolvidos nas estruturas de dados para que possamos utilizá-las melhor e de forma mais eficiente.

Assim, este livro apresentará os conceitos básicos e avançados das principais estruturas de dados com ênfase nos seus aspectos teóricos e práticos, com programas de exemplos e discussão de cada uma de suas funcionalidades. Consequentemente, com os conhecimentos adquiridos o estudante terá uma visão clara e prática das estruturas de dados, podendo adaptá-las em situações reais de sua vida profissional.

Diante de todos os pontos acima descritos, acreditamos que com o estudo atencioso do material aqui presente, você com certeza será um profissional em destaque no mercado de trabalho!

Bons estudos!

1

Introdução

Para podemos aprofundar em todos os aspectos envolvidos em estrutura de dados, necessitamos criar uma base sólida de conhecimentos básicos que serão primordiais para a sequência dos próximos capítulos. Diante disto, estudaremos os diversos tipos de estrutura de dados, como também suas principais características e aplicações.

OBJETIVOS

- Conhecer os conceitos fundamentais contidos nas estruturas de dados;
- Estudar os aspectos que envolvem a implementação das estruturas de dados;
- Discutir sobre os diversos tipos de estrutura de dados;
- Estudar e implementar funções.

1.1 Conceitos fundamentais

A solução de diversos problemas é feito através de desenvolvimento de sistemas de informação (programas). Segundo (LAUDON & LAUDON, 2007) os sistemas de informação são formados por diversos componentes que efetuam a coleta, processamento, armazenamento e distribuição de informações (dados) destinadas ao controle e coordenação de organizações como também o apoio a tomada de decisões. As principais atividades de um sistema são exibidas no diagrama da figura 1.1. A entrada é responsável pela coleta dos fatos brutos (dados) a serem processados. O processamento (programa) tem a finalidade de transformar os dados de entrada através de uma sequência finita e ordenada de passos. O resultado do processamento é apresentado na saída. O *feedback* auxilia nos ajustes dos dados de entrada de acordo com as saídas obtidas.

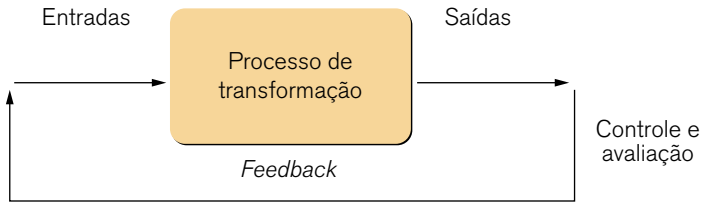


Figura 1.1 – Sistemas de informação.

Levando em conta aspectos citados, têm-se dois pontos principais no desenvolvimento de um programa: dados e procedimentos.

Os procedimentos implementam os aspectos funcionais para o processamento dos dados de um sistema. Diversas técnicas de programação podem ser utilizadas para o desenvolvimento dos procedimentos. Independente da técnica de programação adotada, o objetivo principal do procedimento é tratar os dados. Desta forma, pode-se perceber a importância do conhecimento dos conceitos, organização, manipulação e utilização dos dados para que se desenvolvam programas altamente eficientes.

Para que um problema seja solucionado, os dados referentes ao mesmo devem ser representados como variáveis de entrada, para que o programa possa processá-las. Tomemos como exemplo, um problema onde se deseja calcular o valor de x referente a vela de um pequeno barco, conforme ilustrado na figura1. 2.

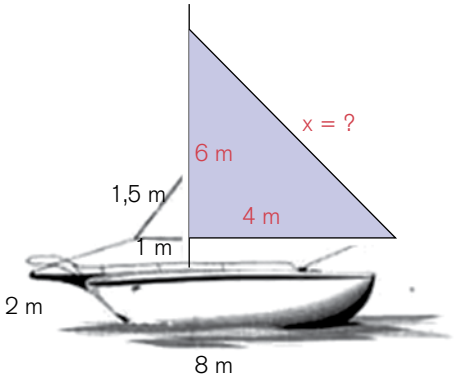


Figura 1.2 – Problema referente a vela de um barco.

Notem que além dos dados da vela, temos outros que não fazem parte do problema. Tanto neste caso, como em outros, deve se selecionar o conjunto de dados que sejam considerados de maior relevância para encontrar os resultados desejados.

Em seguida, define-se o procedimento para efetuar o processamento dos dados selecionados. Como a vela do barco tem a forma de um triângulo retângulo, será utilizado o Teorema de Pitágoras para a resolução do problema. A solução é dada pelo cálculo da hipotenusa através dos valores definidos para os catetos (dados da vela). O programa recebe como dados de entrada, os valores dos catetos, efetua o processamento, calculando a hipotenusa e o resultado é enviado como dado de saída.

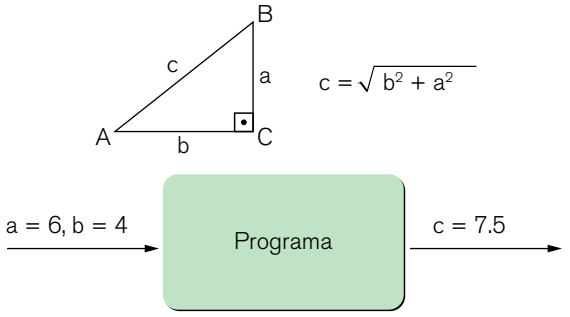


Figura 1.3 – Solução referente a vela de um barco.

Além da escolha dos dados de maior relevância para encontrar os resultados desejados, outro fator importante é de como serão organizados estes dados para a solução de um problema.

Considere um programa para cadastro de cliente de uma concessionária de veículos. Um conjunto de dados relevantes para este cadastro poderia ser: nome, endereço, data de nascimento, etc. Notem que o peso, altura, cor dos olhos e cabelo são dados irrelevantes, para este problema, e desta forma, não fazem parte do cadastro. Se o programa fosse direcionado a uma agência publicitária, estes dados seriam de extrema importância. Isto demonstra que a escolha dos dados deve ter como base o problema a ser resolvido.

Em seguida, definem-se como os dados serão organizados, ou seja, como serão representados um programa. Esta representação está relacionada com as variáveis disponíveis na linguagem que será utilizada no desenvolvimento do programa. A escolha da representação deve levar em conta as operações que serão realizadas sobre os dados. Dependendo da escolha, pode gerar uma maior ou menor complexidade da representação e consequentemente afetará o desempenho do programa. Por exemplo, a operação, imprimir uma lista de cliente, dependendo da complexidade da representação dos dados, pode gerar um programa muito lento. A estrutura de dados diminui sensivelmente a complexidade da representação dos dados, como também tende a criação de programas com maior desempenho.

1.1.1 Conceito de estruturas de dados

Semelhante a um modelo matemático que representa uma realidade física, a estrutura de dados tem como objetivo representar as relações lógicas entre os dados de uma forma coerente para que possam ser processadas e registradas pelo computador.

As estruturas de dados têm um papel importante no desenvolvimento de software permitindo criar programas com uma representação dos dados mais relevante de um problema real, de forma mais clara e limpa. Consequentemente há um ganho elevado no desempenho dos programas. Assim, desenvolver sistemas tendo como centro as estruturas de dados garante a criação de procedimentos mais simples e eficiente melhorando o tempo de execução de um programa, melhora a leitura do código fonte, facilidade de manutenção e diminuição de custos.

De acordo com Mizrahi (2006), uma estrutura de dados pode ser definida como sendo uma coleção de variáveis, podendo ser tipos iguais ou diferentes, reunidas sob um único nome. Vários autores denominam estrutura de dados como sendo registros.

Basicamente as estruturas de dados podem ser divididas em duas formas: homogênea (vetores e matrizes) e heterogênea (registros). As estruturas homogêneas visão armazenar dados de um único tipo, como por exemplo, string ou inteiros. Desta forma, estruturas de dados homogêneas são empregadas em situações onde as informações podem ser organizadas em um único tipo de dados, normalmente utilizando vetores e matrizes. A maioria dos problemas modelados necessita de uma representação dos dados composta por diferentes tipos simultaneamente, tais como, string, inteiros, datas, etc. Esta variação de tipos de dados é permitida na composição de uma estrutura de dados heterogênea.

1.1.2 Tipos de dados, Estrutura de dados e Tipos Abstratos de dados.

Neste ponto é importante entendermos os termos tipos de dados, estrutura de dados e tipos abstratos de dados que aparentemente sejam parecidos, possuem significados diferentes. Os tipos de dados são utilizados pelas linguagens de programação para definir o conjunto de valores que uma variável pode assumir. Por exemplo, o tipo de dado inteiro pode assumir valores como 1, 25, 1896, etc. O tipo data pode assumir valores como “25/12/2015”, “07/09/1822”, etc. Esta visão do tipo de dado esta relacionada ao modo como o computador pode interpretar os dados. Outra forma de visualizar os tipos de dados é em termos do que o usuário deseja fazer com os dados, como por exemplo, somar dois inteiros, ordenar um lista de inteiros, etc. Este tipo de conceito de tipo de dado é conhecido como TAD – Tipo Abstrato de Dado.

Os tipos abstratos de dados são formados por um conjunto de tipos de dados e um conjunto de procedimentos (funções) que podem ser aplicadas sobre este conjunto de tipos de dados. Desta forma, para que um tipo abstrato de dado seja implementado, são utilizados as estruturas de dados. As estruturas de dados representam as relações lógicas entre os dados, por exemplo, relação linear ou não linear (hierárquica) e as operações que serão efetuadas sobre

estes dados, como por exemplo, inserir um cliente no conjunto de dados do cliente, excluir um arquivo de uma árvore de diretórios, etc.

1.1.3 Conceito de função

Durante o desenvolvimento de um programa, acontece de depararmos com algum bloco de código que pode ser utilizado em outras partes do programa. Esta situação nos leva a duas abordagens: copiar o bloco para os locais necessários ou utilizar funções. A primeira abordagem gera uma série de problemas, tais como, manter todas as cópias atualizadas caso haja alguma alteração em uma delas. Muitas cópias do mesmo código pode deixar o programa mais complexo de entender e manter ao longo do tempo. A segunda abordagem, utilizar funções, soluciona ou minimiza ao máximo estes problemas.

A função pode ser definida como um bloco de código com uma tarefa específica. Ou seja, em vez de copiar o bloco de código em várias partes do programa, colocamos o bloco dentro de uma função, e ao longo do programa, quando necessário, chamamos a função que executará o código automaticamente.

De forma geral, os elementos de uma função C++ são:

```
Tipo_do_Dado_de_Retorno Nome_da_Função(Listas de parâmetros...)
{
    Corpo da função
}
```

- **Nome_da_Função**: nome pelo qual a função será chamada para ser executada sua tarefa. O nome da função não pode começar com números.
- **Listas de parâmetros**: são os dados para que a função possa executar sua tarefa.
- **Tipo_do_Dado_de_Retorno**: retorna o resultado da execução da tarefa realizada pela função.
- **Corpo da função**: sequência de comandos para executar a tarde proposta pela função.

1.1.4 Conceito de `struct`

Através de `Struct` é que são definidas as estruturas de dados dentro de C++. Estas estruturas, também conhecidas como registros, são definidas como um

conjunto de variáveis, que são agrupas e acessadas por um único nome dentro do programa. Através de `struct` é possível definir um novo tipo de dado composto por muitas variáveis, denominadas membros da estrutura. O exemplo a seguir, ilustra a utilização de `struct`.

```
struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};
struct DADOS_ALUNO AlunoA;
AlunoA.CodAluno = 10;
strcpy(AlunoA.Nome, "Gabriela");
AlunoA.Turma = 250;
```

Veremos e nos aprofundaremos mais na utilização de struct na implementação de estrutura de danos nos próximos tópicos e capítulos deste livro.

1.1.5 Conceitos de árvore, grafo, pilha, fila e lista

As estruturas de dados podem ser classificadas como: lineares e não lineares. Estruturas do tipo linear são aplicações onde os objetos são representados e manipulados em uma sequência ordenada de valores. Ou seja, a partir de um determinado objeto, há um objeto na sequência. Estruturas do tipo não linear (hierárquico), cada objeto pode ter diferentes objetos na sequência (AGUILAR, 2008).

Estruturas de dados lineares podem ser:

- Pilhas
- Filas
- Lista

Estruturas de dados não lineares podem ser:

- Árvores
- Grafos

1.1.5.1 Listas

As listas são consideradas as estruturas mais simples para interligar elementos de um conjunto. Segundo (VELOSO, 1986) as listas são estruturas que permitem representar um conjunto de dados, que de alguma forma se relacionam, de forma que os elementos fiquem dispostos em sequência (figura 1.4). Cada elemento da lista, por exemplo, x_1, também denominado nó, pode conter um dado primitivo (inteiro, `string`, etc. - figura 1.4) ou dado composto (registro - figura 1.5).

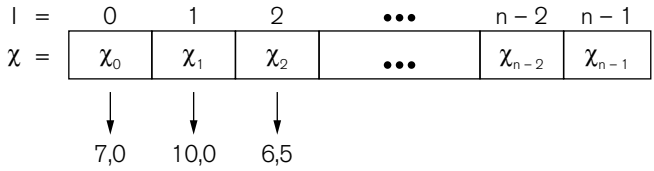


Figura 1.4 – Dado primitivo (Nota de alunos).

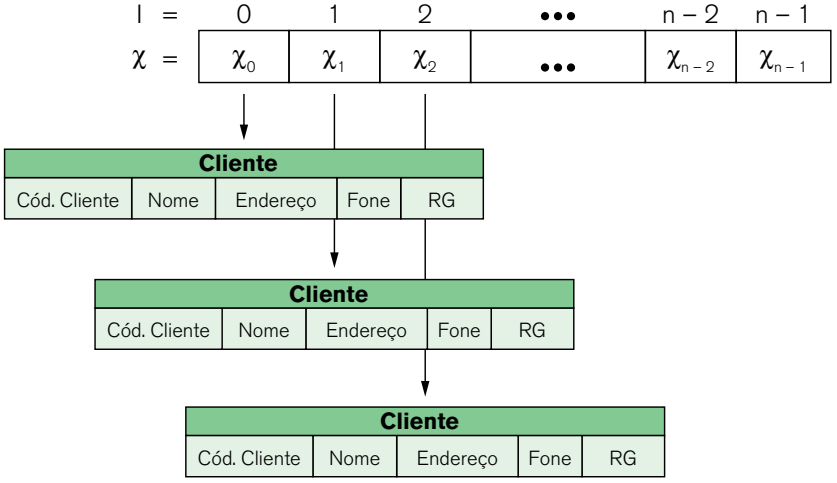


Figura 1.5 – Dado composto (Cadastro de cliente).

As listas podem ser implementadas de forma estática ou dinâmica. No desenvolvimento de um programa pode ser necessário determinar como será a inserção ou a remoção de elementos de uma lista. Dependendo do critério escolhido, poderemos ter uma implementação em forma de pilha ou lista.

As operações implementadas em uma lista dependem do tipo de aplicação.

Algumas operações mais comuns são:

- **Criar**: cria uma lista vazia.
- **Verificar lista vazia**: verifica se há algum elemento na lista.
- **Verificar lista cheia**: verifica se a lista esta cheia.
- **Inserir**: insere um elemento numa determinada posição ou no final da lista.
- **Alterar**: alterar algum elemento da lista.
- **Remover**: remove um elemento de uma determinada posição.
- **Buscar**: acessa um elemento da lista.
- **Exibir a quantidade**: retorna a quantidade de elementos da lista.
- **Combinar**: combina duas ou mais listas em uma única.
- **Dividir lista**: dividi uma lista em duas ou mais.
- **Ordenar**: ordena os elementos da lista de acordo com algum de seus componentes.
- **Esvaziar**: esvaziar a lista.

São exemplos de lista: notas de alunos, cadastro de clientes, produtos em estoque, meses do ano, setores de disco rígido acessado pelo sistema operacional, lista de pacotes transmitido por um computador em uma rede de computadores, etc.

1.1.5.2 Pilhas

Segundo (TANENBAUM, LANGSAM, e AUGENSTEIN, 1995), a pilha é um tipo especial de lista em que os elementos a serem inseridos ou removidos ocorrem no topo da pilha (figura 16). Esta característica é conhecida como LIFO (Last In, First Out - Último a Entrar, Primeiro a Sair). Dessa forma, o último elemento que foi inserido na pilha será o primeiro elemento a ser removido. A estrutura pilha pode ser comparada a uma pilha de pratos ou livros. O prato será sempre colocado no topo da pilha e só poderá ser retirado o prato do topo da pilha, caso contrário a pilha desabaria. Como a pilha é uma derivação das listas, pode ser implementadas de forma estática ou dinâmica.

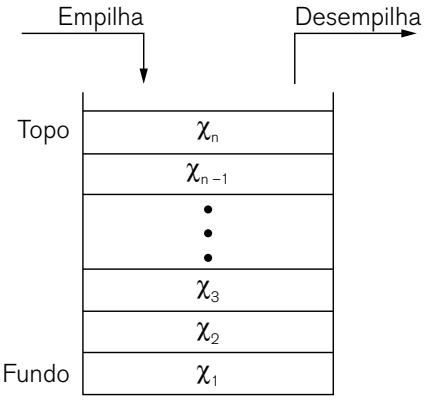


Figura 1.6 – Pilha.

Algumas operações mais comuns em pilha são:

- **Criar**: cria uma pilha vazia.
- **Empilhar**: insere um novo elemento no topo da pilha.
- **Desempilhar**: remove um elemento do topo da pilha.
- **Exibir topo**: exibe o elemento do topo da pilha.
- **Exibir a quantidade**: retorna a quantidade de elementos da pilha.
- **Esvaziar**: esvazia todos os elementos da pilha.

1.1.5.3 Filas.

A fila também é um tipo especial de lista, onde os elementos são inseridos em uma extremidade, chamada início da fila, e retirados na extremidade oposta, chamada final da fila (figura 1.7). Esta característica é conhecida como FIFO (First In, First Out - Primeiro a Entrar, Primeiro a Sair). Desta forma, o primeiro elemento que foi inserido na fila será o primeiro elemento a ser removido. A estrutura fila pode ser comparada a uma fila de banco. O primeiro cliente que chegou, será o primeiro a ser atendido. Também a fila pode ser implementadas de forma estática ou dinâmica.

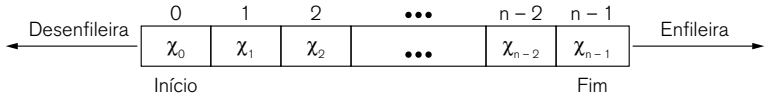


Figura 1.7 – Fila.

As operações mais comuns efetuadas com filas são:

- **Criar**: cria uma fila vazia.
- **Enfileirar**: insere um elemento no fim da fila.
- **Desenfileirar**: remover um elemento no início da fila.
- **Exibir início**: exibe o elemento do início da fila.
- **Exibir a quantidade**: retorna a quantidade de elementos da fila.
- **Esvaziar**: esvazia a fila.

As filas são úteis em diversas aplicações, como por exemplo, os sistemas operacionais, que utilizam filas para gerenciar o escalonamento dos processos que serão executados pelo processador e a alocação de recursos.

1.1.5.4 Árvores

Como visto anteriormente, as filas e pilhas são estruturas de acesso linear e limitam-se a representação de apenas uma dimensão de dados. Diversas aplicações têm como requisito, acesso aos seus dados de uma forma não linear, ou seja, exigem estruturas mais complexas. Para estes casos, destacam-se as árvores, por permitirem uma maior flexibilidade em situações onde exigem uma representação hierárquica de dados. A árvore é definida como sendo um conjunto finito de nós e vértices (figura 18).

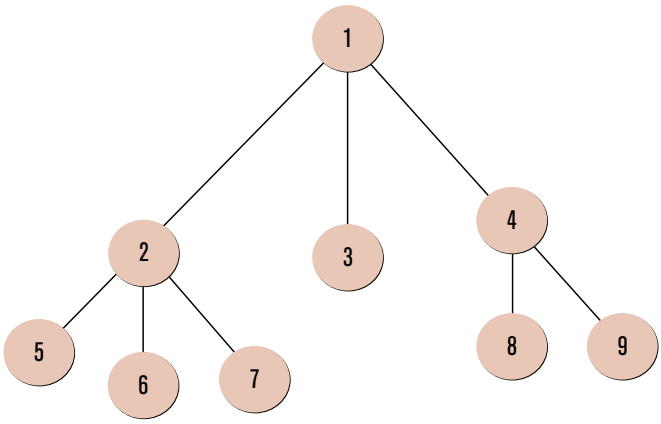


Figura 1.8 – Árvores.

Diferente das árvores naturais, que têm sua origem de baixo para cima, uma estrutura de dados em forma de árvore é representada de cima para baixo. A árvore é composta de nós e arestas (conexões). Os nós, denominados também com vértice, são estruturas de dados que representam as informações a serem processadas. As arestas são as conexões entre os nós. Conexões podem ser do tipo: unidirecional ou bidirecional. O tipo unidirecional pode ir apenas de um nó para outro, mas não pode fazer o caminho contrário. Já o bidirecional pode de qualquer nó chegar a outro. Sendo que o topo é representado pelo nó raiz e os demais nós se conectam a ele, ou a outro que já esteja conectado a ele. O nó raiz é o nó pai dos demais. Um nó pai pode estar conectado a vários nós filhos. Cada nó filho pode estar conectado à apenas um nó pai. A exceção fica por conta do nó raiz que não tem nó pai. Um exemplo de utilização de árvores a estruturação de diretórios de um computador (figura 1.9)

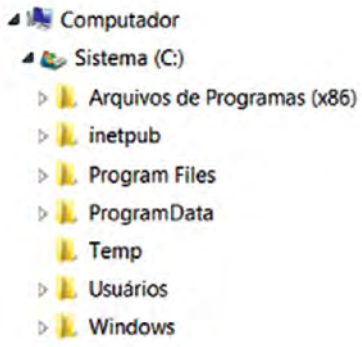


Figura 1.9 – Árvore de diretórios.

CONEXÃO

Leia um pouco mais sobre os conceitos de árvores em: http://www2.dc.ufscar.br/~bsi/materiais/ed/u16.html.

1.1.5.5 Grafos

Os grafos são semelhantes às árvores no sentido matemático. Graficamente também possuem nós e conexões (figura 1.10), mas se difere na programação em relação as árvores.

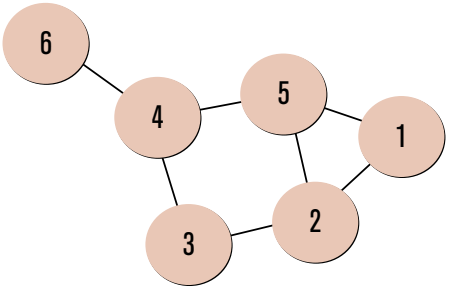


Figura 1.10 – Grafo.

Sua estrutura é formada pela conexão dos nós através das arestas. No caso dos grafos, as arestas podem ou não ser direcionadas. Quando são direcionadas, o grafo é denominado grafo direcionado (figura 1.11). Não há restrições de conexão, assim, o grafo pode ter conexões permitindo ter um caminho de um nó para qualquer outro. Os grafos permitem representar diversos problemas, tais como, redes de computadores, trajetos entres cidades, roteamento de veículos, etc.

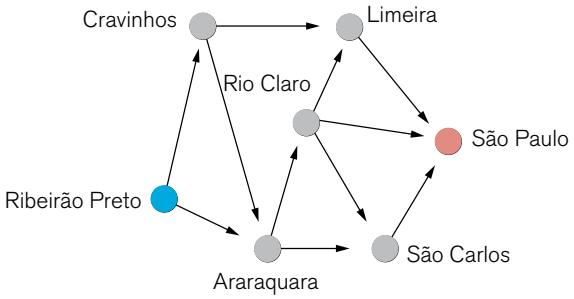


Figura 1.11 – Grafo direcionado. Trajetos entre cidades.

1.1.6 Listas lineares

No desenvolvimento de programas, frequentemente há a necessidade de armazenar um conjunto de nomes, valores, produtos, pessoas etc. Dentre as estruturas de dados, a de maior simplicidade de manipulação para este tipo de tarefa são as listas lineares.

Como mencionado anteriormente, cada nó da lista, pode conter um dado primitivo ou dado composto. Um exemplo de lista linear são os meses do ano: [jan, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dez]. No caso de dados compostos, poderia ser uma lista de dados de cliente e em cada nó teríamos Cliente(Cód Cliente, Nome, endereço, telefone).

As listas lineares são estruturas de dados que têm como objetivo armazenar um conjunto de dados, que de alguma forma se relacionam, com os elementos dispostos em sequência.

1.1.6.1 Definição

Uma lista linear L é uma coleção L:[x1, x2, ..., xn], n>=0 tal que:

- x1 é o primeiro elemento da lista.
- xn é o último elemento da lista.
- xk, x<k<n, é seguido do elemento xk+1 e precedido de xk – 1 na lista.

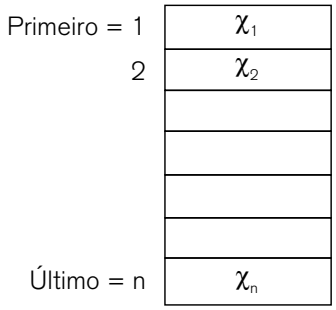


Figura 1.12 – Implementação de uma lista.

Fonte: Ziviani (1996).

As operações realizadas por uma lista podem ser definidas de acordo com a aplicação em que esteja sendo utilizada. Segundo (ZIVIANI, 1996), as operações mais comuns, para as listas, encontradas na maioria das aplicações são:

- **Criar**: cria uma lista vazia.
- **Verificar lista vazia**: verifica se há algum elemento na lista.
- **Verificar lista cheia**: verifica se a lista esta cheia.
- **Inserir**: insere um elemento numa determinada posição ou no final da lista.
- **Alterar**: alterar algum elemento da lista.
- **Remover**: remove um elemento de uma determinada posição.
- **Buscar**: acessa um elemento da lista.
- **Exibir a quantidade**: retorna a quantidade de elementos da lista.
- **Combinar**: combina duas ou mais listas em uma única.
- **Dividir lista**: dividi uma lista em duas ou mais.
- **Ordenar**: ordena os elementos da lista de acordo com algum de seus componentes.

As listas lineares permitem que um elemento possa ser inserido, alterado ou removido em qualquer posição. Assim temos:

- Inserir um elemento na lista:

  O elemento X a ser inserido terá um sucessor e/ou antecessor;

  Para inserir o elemento X na posição p ($1 \leq p \leq n+1$), os elementos x_i são deslocados para a posição x_{i+1}, ($p \leq i \leq n$);

  É acrescido em uma unidade o total de elementos da lista.

Exemplos:

Seja a lista L = [7.0, 8.5, 10, 9.5] representando n = 4 elementos.

$$x_1=7.0; x_2=8.5; x_3=10; x_4=9.5;$$

Para inserir um novo elemento, com valor igual a 6.0 na 3ª posição da lista, os elementos 10 e 9.5 serão deslocados na lista, L = [7.0, 8.5, 6.0, 10, 9.5].

$$x_1=7.0; x_2=8.5; x_3=6.0; x_4=10; x_5=9.5;$$

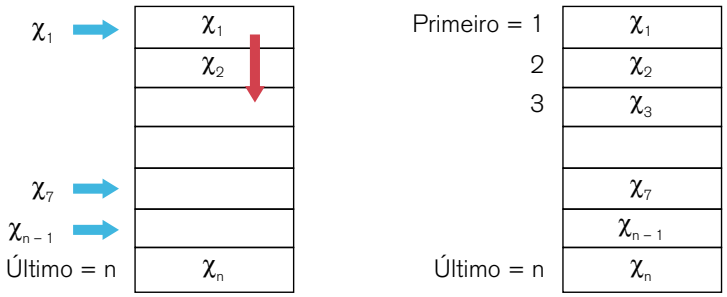


Figura 1.13 – Inserção de elementos na lista linear.

• Buscar elemento da lista

São duas as possibilidades de busca da lista:

– Pode-se encontrar o elemento pela sua posição relativa dentro da lista;

– Ou, o elemento é encontrado pelo seu conteúdo.

• Remover um elemento da lista

Para remover o elemento X na posição p ($1 \leq p \leq n+1$), os elementos x_i são deslocados para a posição x_{i-1}, ($p \leq i \leq n$);

É decrescido em uma unidade o total de elementos da lista.

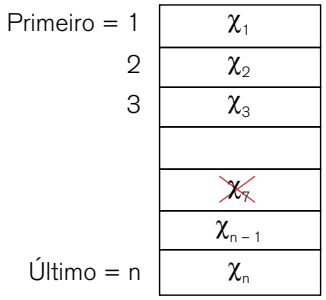


Figura 1.14 – Remoção de elemento da lista.

Exemplos:

Seja a lista L = [7.0, 8.5, 6.0, 10, 9.5] representando n = 5 elementos.

$$x_1=7.0; x_2=8.5; x_3=6.0; x_4=10; x_5=9.5;$$

Para remover o elemento com valor igual a 6.0 na 3ª posição da lista, os elementos 10 e 9.5 serão deslocados na lista, L = [7.0, 8.5, 10, 9.5].

$$x_1=7.0; x_2=8.5; x_3=10; x_4=9.5;$$

1.1.7 Uso de funções definidas pelo programador

Utilizar funções facilita e agiliza o desenvolvimento de programas e promove o reuso de rotinas que podem ser utilizadas, não somente em um programa mais em vários, conforme a necessidade for surgindo. Neste capítulo estudaremos a codificação de funções.

> A função pode ser vista como um atalho para um bloco de código. Isto permite que um programa grande é complexo, seja dividido em blocos menores utilizando funções para representar partes lógicas do processo.

1.1.7.1 Programa Alô Mundo

Para iniciarmos os nossos estudos de funções, faremos um primeiro programa que exibe na tela a frase: Alô Mundo. Os programas em C são inicializados com uma função principal, chamada `main`, ilustrada abaixo. A sintaxe da função `main` pode ter algumas variações que serão vista mais a frente.

```
#include <iostream>
using namespace std;
// Início da Função principal
void main(){
    cout << "Alô mundo";
    system("pause > null");
}
```

A seguir será analisado cada parte do programa acima.

```
#include <iostream>
```

A linguagem C possui um número elevado de bibliotecas para os mais diversos fins. Ao desenvolver um programa, necessitamos de informar ao compilador as bibliotecas que iremos utilizar. Isto é feito através do cabeçalho.

A biblioteca `iostream` possui os comandos de entrada e saída em C++, como por exemplo, enviar dados para o monitor ou receber dados digitado no teclado.

```
using namespace std;
```

Esta linha informa ao compilador que o programa ira utilizar comandos e funções no padrão C++.

```
// ou /* ... */
```

As barras duplas // ou /*...*/ indicam que é um comentário. Comentário são linhas que não serão executadas pelo programa. O comentário serve para fazer anotações importantes dentro do código, e assim, deixa-lo mais legível.

```
void main(){...}
```

Um programa em C++, deve definir ao menos uma função main. A função main indica o ponto inicial da execução do programa. Os blocos de dados ({...}) delimitam todas as linhas de código que um comando ou função deve executar de uma vez.

```
cout << "Alô mundo";
```

O objeto cout representa um fluxo (`stream`) de saída em C++. Neste exemplo, temos um fluxo de dados a ser impresso na tela. O operador <<, denominado operador de inserção, insere dados dentro do `stream`. Quando temos um texto longo e gostaríamos de dividi-los em varias linhas utilizamos `endl`. O código ilustrado abaixo, imprime o texto em duas linhas na tela do computador. A primeira linha

```
cout << "Este é um texto " << endl;
cout << "digitado em duas linhas";

system("pause > null");
```

A função `system(...)` envia comandos para o sistema operacional. Retirando esta linha do código, a janela que mostra o resultado da execução do programa, é aberta e fechada rapidamente, não sendo possível visualizar os resultados. Para que a janela permaneça aberta, utiliza-se o parâmetro `pause > null`. Assim a janela permanecerá aberta até que o usuário aperte qualquer tecla.

1.1.8 Implementação de funções com e sem retorno, com e sem passagem de parâmetros

Iniciaremos o estudo da criação de função, através de um exemplo. A tarefa da função será calcular o perímetro da circunferência. O cálculo do perímetro da circunferência é dado pela fórmula $P = 2 \cdot \pi \cdot r$. O raio será passado para a função como parâmetro, a função calculará e retornará o valor do perímetro da circunferência. A seguir é definido o código para a função.

```
float CalcPerimetroCircunferencia(float raio)
{
    float Perimetro;
    // P = 2.π.r
    Perimetro = 2 * 3.14 * raio ;
    return Perimetro;
}
```

A função foi definida com o nome de CalcPerimetroCircunferencia. Em linguagem C++, há diferença entre letras maiúsculas e minúsculas. Ou seja, uma função com o nome CalcularSalario é diferente de uma chamada CALCULARSALARIO ou calcularsalario. O valor do raio é passado como parâmetro para função. Como o valor do raio pode ter casas decimais o tipo do parâmetro foi definido como `float`. Pelo mesmo motivo, o retorno do resultado da função, foi definido como `float`. O calculo é efetuado e o resultado é armazenado na variável Perimetro. Por fim, a função executa o comando `return` para retornar o valor calculado.

1.1.8.1 Protótipo de função

Antes de utilizar a função, temos que declará-la. A declaração da função é chamada de protótipo da função. O protótipo da função é uma instrução colocada

no início do programa, indicando o tipo da função e os parâmetros que recebem. O objetivo do protótipo é fornecer informações, para checagem de erro, ao compilador. A seguir tem-se o programa completo. Basicamente são todos os elementos da declaração da função mais o ponto e vírgula.

```
float CalcPerimetroCircunferencia(float raio); // Protótipo da função
```

O nome do parâmetro passado para a função não é obrigatório e pode ser omitido, mas o tipo é obrigatório. Dessa forma o protótipo ficaria da seguinte forma:

```
float CalcPerimetroCircunferencia(float); // Protótipo da função
```

Efetuados estes passos, a função esta pronta para ser chamada. A seguir, tem-se o programa completo para melhor visualização dos locais onde cada parte do código é definido.

```
#include <iostream>
using namespace std;

// Protótipo da função
float CalcPerimetroCircunferencia(float raio);
void main(){
    float Per;
    // Chamando a função
    Per = CalcPerimetroCircunferencia(5);
    cout << "O perímetro da circunferência é: " << Per;
    system("pause > null");
}
float CalcPerimetroCircunferencia(float raio)
{
    float Perimetro;
    // P=2.π.r
    Perimetro = 2 * 3.14 * raio ;
    return Perimetro;
}
```

Ao finalizar a execução da função, o resultado retornado pela função é armazenado nesta variável `Per` e em seguida é visualizado na tela através da execução do comando `cout`.

CONEXÃO

Aprofunde seus conhecimentos sobre funções: http://www.tiexpert.net/programacao/c/funcoes.php.

1.1.8.2 Função sem protótipo de função

O protótipo de função pode ser eliminado caso a definição da função seja feita antes de sua chamada. O exemplo anterior ficaria da seguinte forma:

```
#include <iostream>
using namespace std;
float CalcPerimetroCircunferencia(float raio)
{
    float Perimetro;
    // P=2.π.r
    Perimetro = 2 * 3.14 * raio ;
    return Perimetro;
}
void main(){
    float Per;
    // Chama a função a ser executada
    Per = CalcPerimetroCircunferencia(5);
    cout << "O perímetro da circunferência é: " << Per;
    system("pause > null");
}
```

Apesar de ter a possibilidade de não usar o protótipo de função, dentro das boas práticas da programação, é utilizado a declaração do protótipo de função antes de utilizá-la.

1.1.9 Diferença entre parâmetros passados por valor e parâmetros passados por referência

Parâmetros são informações passadas para a função. Uma função pode ter vários valores passados como parâmetros a ser utilizado na função. Na função Empilhar(), ilustrada abaixo, temos 5 parâmetros de entrada da função.

```
bool Empilhar(DADOS_ALUNO Pilha[], int CodAluno, char Nome[],
    int Turma, int &PosTopo) {
    ...
}
```

1.1.9.1 Passagem de parâmetros por valor

Na passagem de parâmetro por valor, a função cria uma cópia dados. A cópia é armazenada em variáveis, que são criadas quando a função é chamada e destruídas quando a função é finalizada. Por exemplo, quando a função Beep(...) descrita a seguir, iniciou sua execução, automaticamente foi criada a variável NroVezes e armazenado o valor 10, passado como parâmetro para a função. Ao final da execução da função, a variável NroVezes é destruída automaticamente.

```
#include <iostream>
using namespace std;
void Beep(int Tempo); // Protótipo da função
void main(){
    Beep(10); // Chama a função a ser executada
}
void Beep(int NroVezes){
    int Ind;
    for(Ind = 0; Ind < NroVezes; Ind++) {
           cout << '\x07';
    }
}
```

Na função CalcPerimetroCircunferencia(...), é retornado um único valor: perímetro da circunferência.

```
float CalcPerimetroCircunferencia(float raio){
    float Perimetro;
    // P=2.π.r
    Perimetro = 2 * 3.14 * raio ;
    return Perimetro;
}
```

O comando `return` permite que apenas um valor seja retornado da função. Para que mais de um valor sejam retornados, é utilizado o conceito de parâmetros por referência. Nas chamadas das funções anteriores, os parâmetros eram passados por valor, ou seja, a função efetuava uma cópia dos valores. Na passagem de parâmetros por referência, a função tem acesso direto às variáveis enviadas. Por exemplo, a função Alterar(...) recebe o valor da variável Idade através de um parâmetro de referência, a variável paramIdade. Note que o tipo de paramIdade está com o símbolo & (int& paramIdade). Isto permite o acesso direto a variável Idade, ou seja, qualquer alteração em paramIdade irá refletir em Idade. Inicialmente a variável Idade tem o valor igual a 20 e é enviada para função. O parâmetro paramIdade recebe o valor 20. Em seguida, paramIdade é alterado para 100. Esta alteração faz com que Idade também tenha o seu valor alterado de 20 para 100.

```
#include <iostream>
using namespace std;
// Protótipo da função
void Alterar(int& paramIdade);
void main(){
    int Idade;
    Idade = 20;
    Alterar(Idade);
    cout << "O valor da idade é: " << Idade << endl;
    system("pause > null");
}
```

```
void Alterar(int& paramIdade){
    paramIdade = 100;
}
```

Os parâmetros de uma função podem conter, juntamente, tanto parâmetros passados por valor como por referência. A próxima função que será criada, terá como objetivo de receber o valor de um produto para ser calculado o valor em dólar e em euros. Como o intuito do exemplo é estudar a passagem por referência e para simplificar a função, o valor do dólar e euro estarão fixos dentro da função.

```
#include <iostream>
using namespace std;
void CalcularPreco(float VlrProdutoReal, float& VlrProdutoDolar,
float& VlrProdutoEuro);
void main(){
    float VlrProdReal, VlrProdDolar, VlrProdEuro;
    cout << "Digite o valor do produto: ";
    cin >> VlrProdReal;
    // Chamada da função
    CalcularPreco(VlrProdReal, VlrProdDolar, VlrProdEuro);
    cout << "Valor em Dolar: " << VlrProdDolar << endl;
    cout << "Valor em Euro: " << VlrProdEuro << endl;
    system("pause > null");
}
void CalcularPreco(float VlrProdutoReal, float& VlrProdutoDolar,
                   float& VlrProdutoEuro){
    VlrProdutoDolar = VlrProdutoReal / 2.621;
    VlrProdutoEuro = VlrProdutoReal / 3.084;
}
```

A função CalcularPreco(...) recebe três parâmetros: VlrProdutoReal, VlrProdutoDolar e VlrProdutoEuro. O parâmetro VlrProdutoReal é passado por valor, enquanto VlrProdutoDolar e VlrProdutoEuro são passados por referência. O valor do produto, ao ser convertido e atribuído às variáveis VlrProdutoDolar e VlrProdutoEuro, automaticamente alteram os valores de VlrProdDolar e VlrProdEuro.

1.1.10 Retorno de valores

1.1.10.1 Tipo de função

O tipo de função é definido de acordo com o tipo de valor que ela retorna. No nosso exemplo, como a função está retornando o valor do tipo `float`, ela é dita do tipo `float`.

```
float CalcPerimetroCircunferencia(...){
...
}
```

Podemos ter situações onde não há necessidade de retornar nenhum valor da função. O exemplo a seguir produz uma quantidade de `beeps`, no alto falante. A quantidade é passada como parâmetro para a função.

```
#include <iostream>
using namespace std;
void Beep(int Tempo); // Protótipo da função
void main(){
    Beep(10); // Chama a função a ser executada
}
void Beep(int NroVezes){
    int Ind;
    for(Ind = 0; Ind < NroVezes; Ind++) {
           cout << '\x07';
    }
}
```

Como a função `Beep(...)`, toca apenas o alto falante e não há nenhum valor de retorno, foi declarado como sendo do tipo `void`. Se a quantidade de `beeps` fosse fixa, por exemplo, dois `beeps`, não haveria a necessidade de passar o parâmetro NroVezes. Assim função não teria parâmetros de entrada e nem valor de retorno. Esta alteração ficaria da seguinte forma.

```
void Beep(){
    int Ind;
    for(Ind = 0; Ind < 2; Ind++) {
           cout << '\x07';
    }
}
```

Alternativamente, poderia ser colocado `void` na declaração de parâmetros.

```
void Beep(void){
    int Ind;
    for(Ind = 0; Ind < 2; Ind++) {
           cout << '\x07';
    }
}
```

1.1.10.2 Comando return

O comando `return` finaliza a execução da função e volta o controle para a instrução após ao da chamada da função. O comando `return` pode ser utilizado de três formas:

```
return;
return expressão;
return (expressão);
```

O `return` sem a expressão, somente pode ser utilizado em função do tipo `void`. Já as duas outras formas podem ser utilizadas das seguintes formas:

```
float CalcPerimetroCircunferencia(float raio){
    // P=2.π.r
    return 2 * 3.14 * raio;
}
```

ou

```
float CalcPerimetroCircunferencia(float raio){
    // P=2.π.r
    return (2 * 3.14 * raio);}
```

1.1.11 Escopo de variáveis (local e global)

A representação dos dados, manipulados por um programa, são feito através de variáveis. Variáveis em C++ são divididos em três categorias fundamentais: integral, flutuante e `void`. Os tipos integrais são variáveis que armazenam números inteiros. Os tipos flutuantes armazenam números com casas decimais. Já `void` são utilizados para descrever conjuntos vazios de valores. `Void` é utilizado basicamente em declarações de funções que não retornam nenhum valor, como visto na seção de funções. A linguagem C++ não permite que nenhuma variável seja declarada do tipo `void`.

Segundo Mizrahi (2006) a linguagem C++ possui cinco tipos fundamentais de variáveis. Os cinco tipos são ilustrados na tabela 1.1 abaixo.

Tipo	Bit	Bytes	Escala
char	8	1	–128 a 127
int	16	2	–32768 a 32767
float	32	4	3.4E-38 a 3.4+38
double	64	8	1.7E-308 a 1.7E+308
void	0	0	nenhum valor

Tabela 1.1 – Tipos fundamentais.

1.1.11.1 Declarando variáveis

O nome de uma variável pode conter a quantidade de caracteres que forem necessários, mas existem algumas regras. Primeiramente, não se pode utilizar números no primeiro caractere, ou seja, o primeiro caractere deve ser uma letra ou um caractere de sublinhado. Segundo, não pode conter palavras reservadas. Na tabela 1.2, são listadas algumas destas palavras reservadas.

asm	double	if	protected	this
auto	else	inline	public	typedef
break	enum	int	register	union
case	extern	interrupt	return	unsigned
catch	far	long	short	virtual
char	float	near	signed	void

class	for	new	sizeof	volatile
const	friend	operator	static	while
default	goto	pascal	switch	
do	huge	private	template	

Tabela 1.2 – Palavras reservadas.. Fonte: Adaptado de Mizrahi (2006).

Ao declarar uma variável é importante utilizar o tipo correto, para que seja reservado o espaço correspondente em memória, para receber o valor do dado.

```
int x;
int y;
float z;

x = 10;
y = 30;
y = 20.5;
```

Variáveis do mesmo tipo como x e y, podem ser declaras juntas, separadas por vírgula.

```
int x, y;
float z;
```

A linguagem C++ permite que as variáveis sejam criadas e inicializadas ao mesmo tempo. Neste caso, cada variável deve ser declarada separadamente.

```
int Idade = 25;
char PesFisicaJuridica = 'F';
float Salario = 4500.50;
```

1.1.12 Implementação de funções tendo vetores como parâmetros

A declaração de matrizes e vetores é feita utilizando colchetes, []. A matriz permite armazenar uma coleção de variáveis do mesmo tipo. O vetor é uma matriz de uma dimensão.

Em C++ é considerada base zero. Base zero significa que o primeiro elemento do vetor inicia na posição zero. No exemplo abaixo, um vetor é criado para receber os valores pagos, por mês, de um determinado produto. O vetor foi criado para receber 12 elementos, sendo o primeiro de índice 0 e o último de índice 11.

```
float PagtoMes [12];
PagtoMes [0] = 180.50;
PagtoMes [1] = 188.50;
PagtoMes [2] = 188.50;
...
PagtoMes [11] = 230.90;
```

O código abaixo cria um vetor para armazenar os códigos de alunos e uma matriz para controlar suas notas bimestrais.

```
// Vetor: Armazena o código do aluno
int CodAluno[15];
// Matriz: Armazena o código do aluno e sua nota bimestral
float NotasAluno [15][4];
CodAluno[0] = 5;
CodAluno[1] = 10;
CodAluno[2] = 15;
NotasAluno[5][0] = 8.7;
NotasAluno[5][1] = 7.5;
NotasAluno[5][2] = 9;
NotasAluno[5][3] = 10;
```

Uma atenção especial deve ser dada a valores do tipo texto. O armazenamento de texto é feito através de vetores do tipo `char`. A linguagem C++ não permite que seja atribuído diretamente um texto à variável. Caso isso seja feito, o compilador irá gerar uma mensagem de erro. A atribuição de texto é feita utilizando a função `strcpy`(destino, origem). A função `strcpy`(...) copia o texto do parâmetro origem para uma variável colocada no parâmetro destino.

```
char NomeDisciplina[100];
NomeDisciplina = "Estrutura de Dados"; // ERRADO: NÃO PERMITIDO
strcpy( NomeDisciplina, "Estrutura de Dados"); // CERTO: Permitido
```

Caso o valor da variável é um texto que o usuário deva digitar, usa-se o `cin`. O objeto `cin` representa o `stream` de entrada, ou seja, ele efetua a leitura dos dados digitados no teclado e armazena na variável.

```
#include <iostream>
using namespace std;
void main(){
    system("chcp 1252 > nul");
    char NomeAluno[100];
    cout << "Digite o nome do aluno: ";
    cin >> NomeAluno;
    system("pause > null");
}
```

Veremos mais a frente, nos capítulos sobre implementação de pilha, fila, etc., que muitas vezes há necessidade de copiar um vetor para outro. Isto é feito da seguinte forma:

```
#include <iostream>
using namespace std;
#define TAM_MAX 10
void main(){
    int Ind;
    double CodAlunos[TAM_MAX], CopiaCodAlunos[TAM_MAX];
    for(Ind=0; Ind < TAM_MAX; Ind++){
            // Copia os dados de um vetor para outro
            CopiaCodAlunos[Ind] = CodAlunos[Ind];
    }
    system("pause > null");
}
```

1.1.12.1 Passando vetor para funções

O programa a seguir demonstrar a passagem de vetores como parâmetros de funções. O programa cria uma função que tem como objetivo armazenar os valores pagos de um produto no vetor PagtoMes[12]. A função recebe como parâmetros, o vetor e o valor para ser atribuído a cada mês.

```
#include <iostream>
using namespace std;
void GerarValores(float PagtoM[], float VlrPagto);
void main(){
    int Ind;
    float PagtoMes[12];
    GerarValores(PagtoMes, 150.80);
    for(Ind = 0; Ind < 12; Ind++){
            cout << "Pagamento Mês " << Ind << ": " << PagtoMes[Ind]
<< endl;
    }
    system("pause > null");
}
void GerarValores(float PagtoM[], float VlrPagto){
    int Ind;
    for(Ind = 0; Ind < 12; Ind++){
            PagtoM[Ind] = VlrPagto;
    }
}
```

A função GerarValores(...), recebe o vetor através do parâmetro, float PagtoM[]. O vetor é passado para a função através do seu nome, PagtoMes.

```
GerarValores(PagtoMes, 150.80);
```

O nome de um vetor indica o endereço do seu primeiro elemento. Quando é passado PagtoMes para a função, esta sendo passado o endereço do seu primeiro elemento. Dessa forma, a função pode acessar diretamente os seus elementos. Isto significa que um vetor ou matriz, sempre é passado como referência para uma função. Os elementos que forem alterados no vetor PagtoM[] automaticamente serão alterado no vetor PagtoMes[].

1.1.12.2 Escopo

A utilização de uma variável dependerá do seu escopo. O escopo de uma variável e o conjunto de regras de utilização desta variável. O escopo das variáveis em C++ pode ser dividido em:

Variáveis locais

São variáveis declaradas dentro de uma função. Estas variáveis somente podem ser acessadas dentro da função, ou seja, outras funções não tem acesso a elas.

```
#include <iostream>
using namespace std;
void main(){
    int ValorA, ValorB, Total;
    cout << "Digite o valor de A: ";
    cin >> ValorA;
    cout << "Digite o valor de B: ";
    cin >> ValorB;
    Total = ValorA + ValorB;
    cout << "Total = " << Total;
    system("pause > null");
}
```

As variáveis ValorA, ValorB e Total são declaradas localmente dentro do bloco da função `main()`. Assim, somente dentro da função `main()` poderão ser acessadas. Para que, outras funções tenham acesso as variáveis ValorA, ValorB, estas devem ser declaradas como globais

Variáveis globais

São variáveis declaradas fora de qualquer função. Dessa forma, podem ser acessadas e alteradas por qualquer função dentro do programa.

Para ilustrar, o programa anterior será alterado para que o escopo das variáveis ValorA, ValorB sejam globais.

```
#include <iostream>
using namespace std;
int Somar();
int ValorA, ValorB;
```

```
void main(){
    int Total;
    cout << "Digite o valor de A: ";
    cin >> ValorA;
    cout << "Digite o valor de B: ";
    cin >> ValorB;
    Total = Somar();
    cout << "Total = " << Total;
    system("pause > null");
}
int Somar(){
    int ValorTotal;
    ValorTotal = ValorA + ValorB;
    return ValorTotal;
}
```

Note que as variáveis ValorA e ValorB foram declaradas fora de qualquer função. Dessa forma, tanto a função `main()` quanto `Somar()`, puderam acessá-las.

CONEXÃO

Aprofunde seus conhecimentos sobre variáveis locais e globais: http://www.tiexpert.net/programacao/c/variaveis-globais-e-locais.php.

1.1.13 Construção de biblioteca de funções

Durante o projeto de aplicativos, são desenvolvidas diversas funções que podem ser reaproveitadas no mesmo ou em outro projeto. Por exemplo, é comum em um programa, ser efetuado operações de somar, subtrair, multiplicar e dividir. Estas quatro operações poderiam fazer parte de uma biblioteca de funções e ser utilizadas no programa.

O exemplo a seguir, ilustra um exemplo onde é criada uma biblioteca de funções com as quatro operações matemáticas.

```
#include <iostream>
using namespace std;
int Somar();
int Subtrair();
int Multiplicar();
int Dividir();
int ValorA, ValorB;
void main(){
    int Total;
    cout << "Digite o valor de A: ";
    cin >> ValorA;
    cout << "Digite o valor de B: ";
    cin >> ValorB;
    Total = Somar();
    cout << "Total = " << Total;
    system("pause > null");
}
int Somar(){
    int ValorTotal;
    ValorTotal = ValorA + ValorB;
    return ValorTotal;
}
int Subtrair(){
    int ValorTotal;
    ValorTotal = ValorA - ValorB;
    return ValorTotal;
}
int Multiplicar (){
    int ValorTotal;
    ValorTotal = ValorA * ValorB;
    return ValorTotal;
}
int Dividir (){
    int ValorTotal;
    ValorTotal = ValorA / ValorB;
    return ValorTotal;
}
```

ATIVIDADES

01. De acordo com o material, assinale a opção correta em relação a estrutura de dados pilha.

a) O primeiro elemento a ser inserido será o primeiro a ser retirado.
b) O primeiro elemento a ser inserido irá até o fim da sequencia e será retirado.
c) O elemento é retirado da base da pilha.
d) O último elemento a ser inserido será o primeiro elemento a ser retirado.
e) O último elemento a ser inserido será o último elemento a ser retirado.

02. Qual a diferença entre pilha e fila?

03. De acordo com o material qual a melhor estrutura de dados para representar problemas do tipo redes de computadores, trajetos entres cidades, roteamento de veículos, etc.?

a) lista
b) pilha
c) fila
d) árvore
e) grafo

04. Analise o código a seguir e responda qual a saída do programa.

```
int main () {
    int y = 3;
    cout << y << endl;
    funcaoZ (y);
    cout << y << endl;
}
void funcaoZ (int z) {
    int x;
    x = z;
    cout << z << endl;
    z = 5 ;
    cout << x << endl;
}
```

a) 3 3 5 5
b) 3 3 3 5
c) 3 3 3 3
d) 3 3 0 5
e) 3 3 5 0

05. Um programador esta desenvolvendo uma estrutura de dados para um sistema de cadastro de veículo. Defina a estrutura (registro) para conter os seguintes dados: modelo do veículo (com 100 caracteres), marca (50 caracteres), ano de fabricação (número) e ano do modelo (número).

REFLEXÃO

Neste capítulo aprendemos de forma geral, as características e funções das estruturas de dados, que servirá como base para o aprofundamento na sequência de nossos estudos. Estudamos conceitos envolvidos na elaboração das estruturas de dados, tais como, tipos de dados e tipos abstratos de dados.

Vimos como são utilizadas as funções e variáveis em C++. Iniciamos como estudo da principal função em C++, a função `main()`. Posteriormente vimos como criar nossas próprias funções. Estudamos as formas de enviar as variáveis como parâmetros por valor e por referência para que as funções possam processá-las.

Sugerimos que você faça todos os exercícios propostos e pesquise outras fontes para aprofundar seus conhecimentos. Em caso de dúvidas, retorne aos tópicos e faça a releitura com bastante atenção.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de estrutura de dados e demais assuntos deste capítulo, consulte a sugestão de links abaixo: CELES, W.; RANGEL, J. L. Apostila de Estruturas de Dados. Disponível em: http://www-usr.inf.ufsm.br/~juvizzotto/elc1067-2013b/estrut-dados-pucrio.pdf. Acesso em: Jan: 2015.

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de sistemas operacionais e demais assuntos deste capítulo, consulte as sugestões de links abaixo:

Capítulo 1, 5 do livro: MIZRAHI, V. V. Treinamento em Linguagem C++: Módulo 1. 2ª ed. São Paulo, Editora Prentice-Hall, 2006.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILAR, L. J. **Fundamentos de Programação: Algoritmos, Estruturas de Dados e Objetos.** 3ª ed. São Paulo: McGraw-Hill, 2008.

CELES, W.; RANGEL, J. L. **Apostila de Estruturas de dados.** Disponível em: http://www-usr.inf.ufsm.br/~juvizzotto/elc1067-2013b/estrut-dados-pucrio.pdf. Acesso em: Jan: 2015.

DEITEL, H. M., & DEITEL, P. J. **C++ Como Programar.** 3ª ed. Porto Alegre: Bookman, 2001.

LAUDON, K. C., & LAUDON, J. P. **Sistemas de Informação Gerenciais.** 6ª ed. São Paulo: Prentice Hall, 2007.

MIZRAHI, V. V. **Treinamento em Linguagem C++: Módulos 1 e 2.** 2ª ed. São Paulo: Prentice Hall, 2006.

SCHILDT, H. **C Completo e Total.** 3ª ed. São Paulo: Makron Books Editora Ltda, 1996.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos.** 2ª ed. Rio de Janeiro: LTC, 1994.

TANENBAUM, A. M., LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de Dados Usando C.** São Paulo: Makron Books, 1995.

VELOSO, P. E. **Estruturas de Dados.** 4ª ed. Rio de Janeiro: Campus, 1986.

VILLAS, M. E. **Estruturas de Dados: conceitos e técnicas de implementação.** Rio de Janeiro: Campus, 1993.

2

Estruturas Heterogêneas, Ordenação e Pesquisa

Neste capítulo, iremos estudar as estruturas heterogêneas e como podemos programa-las em C++. Dessa forma iremos entender a importância destas estruturas na implementação de diversos aplicativos, principalmente os que utilização estruturas de dados.

Em seguida veremos o uso de métodos de ordenação. O conhecimento de como funcionam e como implementar um algoritmo de ordenação é fundamental no desenvolvimento de aplicativos que manipulam dados. Veremos diversos algoritmos de ordenação, tais como, `insertion sort`, `selection sort` e `bubble sort`.

Finalizaremos o capítulo, estudando os métodos de pesquisa sequencial e binária de dados.

OBJETIVOS

- Definição e declaração de estruturas heterogêneas localmente e globalmente;
- Uso das estruturas heterogêneas definidas pelo programador;
- Compreender e usar os diversos métodos de ordenação;
- Compreender e usar os métodos de pesquisa sequencial e binária.

2.1 Uso das estruturas heterogêneas definidas pelo programador

Como vimos anteriormente, a maioria dos problemas modelados necessitam de uma representação dos dados composta por diferentes tipos simultaneamente, tais como, `string`, inteiros, datas, etc. Esta variação de tipos de dados é permitida na composição de uma estrutura de dados heterogênea.

As estruturas de dados, ou como são chamadas também de registros, são definidas através de `struct`. Em C++, utilizando struct é possível definir um novo tipo de dado composto por várias variáveis, denominadas membros da estrutura.

O exemplo a seguir, ilustra a utilização de `struct`.

```
struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};
```

CONEXÃO

Para entender melhor TAD – Tipo Abstrato de Dado, veja o link: http://www2.dc.ufscar.br/~bsi/materiais/ed/u2.html. O autor ilustra o conceito de TAD utilizando vários exemplos do cotidiano e do desenvolvimento de sistemas.

2.2 Tipos de elementos que podem ser membros de uma estrutura

Basicamente todos os tipos de variáveis em C++ podem ser utilizadas como membros de uma estrutura. Os membros mais comuns são: `int`, `char`, `float` etc.

2.3 Definição e declaração de estruturas heterogêneas localmente e globalmente

No exemplo a seguir, a estrutura heterogênea foi declarada localmente dentro da função `main()`. Isto implica que a estrutura somente poderá ser utilizada dentro da função `main()`.

```
#include <iostream>
using namespace std;

void main(){
    struct DADOS_ALUNO{
           int CodAluno;
           char Nome[100];
           int Turma;
    };

    struct DADOS_ALUNO AlunoA;

    AlunoA.CodAluno = 10;
    strcpy(AlunoA.Nome, "Gabriela");
    AlunoA.Turma = 250;

    cout << "Código do Aluno: " << AlunoA.CodAluno << endl;
    cout << "Nome: " << AlunoA.Nome << endl;
    cout << "Turma: " <<  AlunoA.Turma << endl;

    system("pause > null");
}
```

Inicialmente foi definida uma estrutura chamada DADOS_ALUNO com três variáveis membros: CodAluno, Nome e Turma.

```
struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};
```

Uma vez a estrutura definida, é possível declarar variáveis do tipo DADOS_ALUNO.

```
struct DADOS_ALUNO AlunoA;
```

Os membros da estrutura podem ser acessados através do operador ponto. O programa atribui valores aos membros da estrutura e em seguida exibiu os valores na tela.

Um ponto a ser observado neste trecho é em relação ao membro AlunoA. Nome. Por ser um vetor do tipo `char`, não é possível lhe atribuir um texto diretamente. Neste caso é utilizado a função `strcpy`(...). A função `strcpy`(destino, origem) copia um texto origem para dentro de uma variável destino.

```
AlunoA.CodAluno = 10;
strcpy(AlunoA.Nome, "Gabriela");
AlunoA.Turma = 250;

cout << "Código do Aluno: " << AlunoA.CodAluno << endl;
cout << "Nome: " << AlunoA.Nome << endl;
cout << "Turma: " <<  AlunoA.Turma << endl;
```

Para que possa ser utilizada por todas as funções que possam ser criadas dentro da aplicação, a estrutura deve ser declarada globalmente, ou seja, fora da função `main()`.

```
#include <iostream>
using namespace std;

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};
struct DADOS_ALUNO AlunoA;
```

```
void main(){
    AlunoA.CodAluno = 10;
    strcpy(AlunoA.Nome, "Gabriela");
    AlunoA.Turma = 250;

    cout << "Código do Aluno: " << AlunoA.CodAluno << endl;
    cout << "Nome: " << AlunoA.Nome << endl;
    cout << "Turma: " <<  AlunoA.Turma << endl;

    system("pause > null");
}
```

2.4 Construção de funções usando estruturas heterogêneas

As estruturas podem ser passadas para funções da mesma forma como são feitas com as variáveis. O tipo do parâmetro da função, que recebe a estrutura, deve ser do mesmo tipo desta estrutura.

No exemplo a seguir a variável AlunoA foi passada para a função Imprimir(...). Notem que o tipo do parâmetro que recebe a variável é do tipo da estrutura DADOS_ALUNO.

```
#include <iostream>
using namespace std;

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};
```

```
void Imprimir(DADOS_ALUNO Aluno);

void main(){
    struct DADOS_ALUNO AlunoA;

    cout << "Digite o código do aluno: ";
    cin >> AlunoA.CodAluno;

    cout << "Digite o nome do aluno: ";
    cin >> AlunoA.Nome;

    cout << "Digite a turma: ";
    cin >> AlunoA.Turma;

    // Envia a estrutura para ser impressa na tela
    Imprimir(AlunoA);

    system("pause > null");
}
void Imprimir(DADOS_ALUNO Aluno){
    cout << "Código do Aluno: " << Aluno.CodAluno << endl;
           cout << "Nome: " << Aluno.Nome << endl;
           cout << "Turma: " <<  Aluno.Turma << endl;
}
```

2.5 Implementação de programas usando estruturas heterogêneas

Veremos nos próximos tópicos e capítulos a utilização de estruturas heterogêneas na construção de diversos programas. Inicialmente veremos a construção de programas de ordenação, tais como `insertion sort`, (inserção), `selection sort` (seleção), `bubble sort` (bolha) etc.

Em seguida, construiremos diversos programas voltados especificamente para os algoritmos de estruturas de dados, tais como, lista, fila, pilha, etc.

2.6 Ordenação

A ordenação é um dos requisitos mais comuns em aplicações. Um simples relatório, produzido por um programa, terá um ordenação levando em conta algum critério. Para que os dados sejam ordenados, podem-se adotar duas abordagens: ao inserir um determinado elemento na lista, respeitar a ordenação da estrutura, ou aplicar algum algoritmo de ordenação a um conjunto de dados já criado. Uma vez que um conjunto de dados esteja ordenado, há uma grande facilidade na recuperação de um determinado elemento deste conjunto. Todos estes aspectos mostram a importância de conhecer os algoritmos de ordenação para melhor aplica-los ás estrutura de dados.

2.6.1 Métodos de ordenação

Diversos são os métodos de ordenação. Todos têm suas vantagens e desvantagens. Vários são os fatores que influência no desempenho de um algoritmo de ordenação, tais como: a quantidade de dados a serem ordenados, se todos os dados caberão na memória interna disponível, forma utilizada pelo algoritmo para ordenar os dados, etc.

Podemos citar, como os métodos de ordenação mais importantes: Bubble Sort (ou ordenação por flutuação), Heap Sort (ou ordenação por heap), Insertion Sort (ou ordenação por inserção), Merge Sort (ou ordenação por mistura) e o Quicksort.

CONEXÃO

Leia um pouco mais sobre ordenação em: http://www.ft.unicamp.br/liag/programacao/ordenacao.html. Entenda um pouco mais com este outro artigo: http://www.decom.ufop.br/menotti/aedl082/tps/tp3-sol1.pdf.

2.7 Compreender e usar o método de ordenação insertion sort, (inserção) em estruturas homogêneas e em estruturas heterogêneas.

O método de ordenação por inserção é muito simples e eficiente. Funciona semelhante a ordenar cartas de um baralho. Ao pegar uma carta, compara o seu valor com as que estão nas mãos. Em seguida, coloca a carta no local correto. A ordenação por inserção é ideal para listas pequenas. O algoritmo percorre o vetor da esquerda para a direita. Conforme vai avançando, os elementos mais a esquerda vão ficando ordenados (figura 2.1).

A figura ilustra a ordenação utilizando uma estrutura homogênea (vetor). Inicialmente verifica se o número 8 é menor que 5, como não é, não efetua a troca. Em seguida verifica se 7 é menor que 8 e 5. Como é menor apenas que 8, então 7 e 8 trocam de posição. Verifica se 4 é menor que 8, 7 e 5. Como ele é menor que 5, então 8 ocupa a posição de 4, 7 ocupa a posição de 8, 5 ocupa a de 7, deixando sua posição antiga vazia. Então 4 vai para posição deixada pelo 5. Este procedimento é executado sucessivamente até que o vetor esteja ordenado.

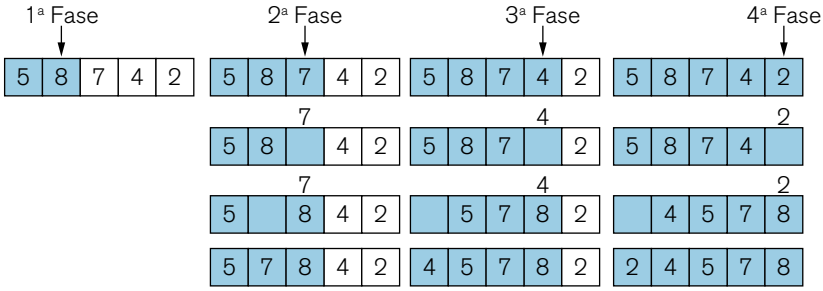


Figura 2.1 – Inserção.

O código do `Insert sort` é definido a seguir para uma estrutura heterogênea. A função recebe o vetor a ser ordenado, é a posição do seu último elemento.

```
bool InsertionSort(DADOS_ALUNO Alunos[], int Pos) {
    DADOS_ALUNO eleito;
    int i, j;

    if (Pos == 0){
           cout << "ERRO: Vetor vazio.";
           return false;
  }
  for (i = 1; i < Pos; i++){
    eleito = Alunos[i];
    j = i - 1;
    while ((j>=0) && (eleito.CodAluno < Alunos[j].CodAluno)) {
           Alunos[j+1] = Alunos[j];
           j--;
     }
     Alunos[j+1] = eleito;
  }
           return true;
}
```

2.8 Compreender e usar o método de ordenação selection sort (seleção) em estruturas homogêneas e em estruturas heterogêneas.

O algoritmo de ordenação por seleção percorre o vetor e coloca na primeira posição o menor valor. Em seguida, varre novamente o vetor, partindo da segunda posição. Encontrando o menor valor, este é colocado na segunda posição. O procedimento é executado sucessivamente até que o vetor esteja ordenado. A figura ilustra a ordenação utilizando uma estrutura homogênea (vetor).

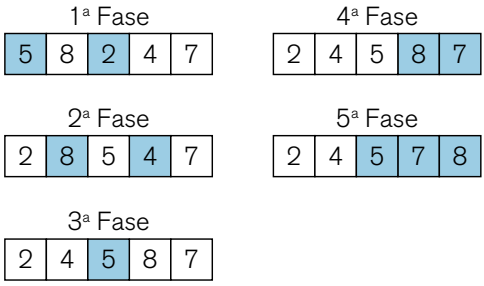


Figura 2.2 – Seleção.

O código do `Selection sort` é definido a seguir para uma estrutura heterogênea. A função recebe o vetor a ser ordenado, é a posição do seu último elemento.

```
bool  SelectionSort(DADOS_ALUNO Alunos[], int Pos)  {
  DADOS_ALUNO aux;
  int i, j, min;
  if (Pos == 0){
    cout << "ERRO: Vetor vazio.";
    return false;
  }
  for (i = 0; i < (Pos-1); i++)
   {
    min = i;
    for (j = (i+1); j < Pos; j++) {
      if(Alunos[j].CodAluno < Alunos[min].CodAluno) {
        min = j;
      }
    }
    if (i != min) {
      aux = Alunos[i];
      Alunos[i] = Alunos[min];
      Alunos[min] = aux;
    }
  }
      return true;
}
```

2.9 Compreender e usar o método de ordenação bubble sort (bolha) em estruturas homogêneas e em estruturas heterogêneas

O `Bubblesort` é um dos métodos de ordenação mais conhecidos e de fácil implementação. A ordenação, utilizando este método, é feita através de troca de valores entre posições consecutivas (figura 2.3). A figura ilustra a ordenação utilizando uma estrutura homogênea (vetor). Neste algoritimo, um determinado valor é levado para posições mais altas ou mais baixas do conjunto de valores. Dado o exemplo da figura 2.3, com os valores iniciais, será aplicado o `Bublesort`. Inicialmente comparece os dois primeiro elementos do vetor: 7 e 5. Como o valor 7 é maior que 5, então, trocam de posição. Em seguida compara 7 com 9. Como esta ordenados, não há troca. Por fim, a última comparação é feita a troca entre 9 e 3. Na próxima fase, o 9 não será comparado. A comparação irá até o elemento anterior ao 9. Este processo é repetido até que todo o conjunto esteja ordenado.

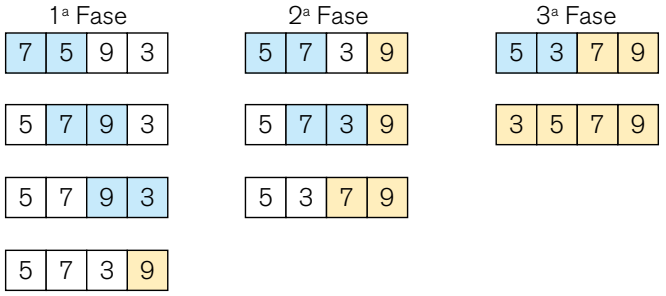


Figura 2.3 - Bublesort.

> O termo **`Bubble Sort`** (literalmente "flutuação por Bolha") vem da forma como as bolhas se comportam em um tanque. Dentro de um tanque, as bolhas se arranjam de forma a procurarem o próprio nível.

O código do `Bubblesort` é definido a seguir para uma estrutura heterogênea. A função recebe o vetor a ser ordenado, é a posição do seu último elemento.

```
bool BubbleSort(DADOS_ALUNO Alunos[], int Pos) {
  DADOS_ALUNO Aux;
  int i, j;

  if (Pos == 0){
    cout << "ERRO: Vetor vazio.";
    return false;
  }

  for(j=Pos-1; j>=1; j--)
  {
    for(i=0; i<j; i++)
    {
      if(Alunos[i].CodAluno > Alunos[i+1].CodAluno)
      {
        Aux = Alunos[i];
        Alunos[i] = Alunos[i+1];
        Alunos[i+1]= Aux;
      }
    }
  }
        return true;
}
```

2.10 Pesquisa

Como na ordenação, a pesquisa é encontrada na maioria de sistemas computacionais. Por exemplo, um programa de controle de matrícula de uma universidade, pode localizar os dados de um aluno através do seu número de matrícula, nome ou CEP. Caso a base de dados tenha milhares de alunos cadastrados, o algoritmo de busca deve ser muito eficiente para que não haja demora no retorno das informações do aluno pesquisado. Este pequeno exemplo mostra a importância do estudo dos métodos de pesquisa.

2.10.1 Compreender e usar os métodos de pesquisa sequencial em estruturas homogêneas e em estruturas heterogêneas.

A pesquisa sequencial é a mais simples de localizar um determinado elemento em um vetor (estruturas homogêneas). Isto porque o algoritmo percorre o vetor, comparando o elemento de interesse com cada elemento do vetor até que encontre o elemento procurado.

O código da pesquisa sequencial é definido a seguir. A função recebe o vetor de registros (estruturas heterogênea), o código do aluno a ser pesquisado, a posição do último elemento do vetor e a variável que receberá os dados do aluno, caso exista no vetor.

```
bool PesquisaSequencial(DADOS_ALUNO Alunos[], int CodAluno, int Pos,
     DADOS_ALUNO &AlunoPesq) {
     int ind;
     if (Pos == 0){
            cout << "ERRO: Vetor vazio.";
            return false;
    }
    for(ind = 0; ind < Pos; ind++ ){
            if(CodAluno == Alunos[ind].CodAluno) {
                   AlunoPesq = Alunos[ind];
                   return true;
        }
    }
    return false;
}
```

2.10.2 Compreender e usar os métodos de pesquisa binária em estruturas homogêneas e em estruturas heterogêneas

Para que seja utilizada a pesquisa binária, o vetor deve estar ordenado. A ideia é localizar o elemento central do vetor e compará-lo ao elemento procurado. Caso o elemento central for maior que o elemento procurado, então a próxima procura será na segunda parte do vetor (estruturas homogêneas) (figura 2.4). Por exemplo, deseja-se localizar o elemento com valor igual a 36. Inicialmente localiza-se o valor central do vetor, no caso o valor 11. Como 11 é menor que 36,

a próxima busca será na segunda parte do vetor. Divide-se a segunda parte e localiza o valor central, no caso o 36. Compara-se o elemento central ao elemento procurado. Como neste caso foi encontrado, então para-se as busca, caso contrário, continuaria o procedimento até localizar o elemento procurado.

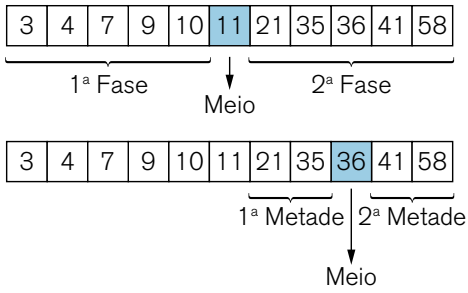


Figura 2.4 – Busca binária.

O código da pesquisa binária é definido a seguir para uma estrutura heterogênea. A função recebe o vetor dos dados, o código do aluno a ser pesquisado, a posição do último elemento do vetor e a variável que receberá os dados do aluno, caso exista no vetor.

```
   bool PesquisaBinaria (DADOS_ALUNO Alunos[], int CodAluno, int Pos,
DADOS_ALUNO &AlunoPesq)
   {
      int inf = 0;           //Limite inferior
      int sup = Pos-1;       //Limite superior
      int meio;
      if (Pos == 0){
       cout << "ERRO: Vetor vazio.";
       return false;
      }
       while (inf <= sup)
      {
             meio = (inf + sup)/2;
              if (CodAluno == Alunos[meio].CodAluno){
                     AlunoPesq = Alunos[meio];
                     return true;
   }
```

```
            else if (CodAluno < Alunos[meio].CodAluno)
                sup = meio-1;
            else
                inf = meio+1;
        }
        return false;   // não encontrado
    }
```

ATIVIDADES

01. Ao inserir um valor em uma lista sequencial ordenada, o que acontecerá com esta lista?

a) A ordem da lista será altera e a quantidade de elementos será alterada.
b) Será feita uma busca com divisões sucessivas da lista.
c) A ordem da lista será alterada para decrescente.
d) A ordem da lista será alterada para crescente.
e) A ordem da lista será mantida e a quantidade de elementos será alterada.

02. Qual tipo de busca utiliza a técnica de divisão, sucessivas, ao meio da lista para encontrar um determinado valor pesquisado?

a) Pesquisa sequencial
b) Inserção
c) Bolha
d) Pesquisa binária
e) Seleção

REFLEXÃO

Neste capítulo aprendemos as características e funções das estruturas de dados, que servirá como base para o aprofundamento na sequência de nossos estudos. Para tanto, vimos definições das estruturas de dados, seus objetivos e suas formas homogênea e heterogênea. Estudamos conceitos envolvidos na elaboração das estruturas de dados, tais como, tipos de dados e tipos abstratos de dados.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de ordenação e demais assuntos deste capítulo, consulte a sugestão de link abaixo:

NAZARÉ JÚNIOR, A. C.; GOMES, D. M. ALGORITMOS E ESTRUTURAS DE DADOS: Métodos de ordenação interna. Disponível em: http://www.decom.ufop.br/menotti/aedl082/tps/tp3-sol1.pdf. Acesso em: Jan: 2015.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILAR, L. J. **Fundamentos de Programação: Algoritmos, Estruturas de Dados e Objetos.** 3ª ed. São Paulo: McGraw-Hill, 2008.

CELES, W.; RANGEL, J. L. **Apostila de Estruturas de dados.** Disponível em: http://www-usr.inf.ufsm.br/~juvizzotto/elc1067-2013b/estrut-dados-pucrio.pdf. Acesso em: Jan: 2015.

DEITEL, H. M., & DEITEL, P. J. **C++ Como Programar.** 3ª ed. Porto Alegre: Bookman, 2001.

LAUDON, K. C., & LAUDON, J. P. **Sistemas de Informação Gerenciais.** 6ª ed. São Paulo: Prentice Hall, 2007.

MIZRAHI, V. V. **Treinamento em Linguagem C++**: Módulo 1 e 2. 2ª ed. São Paulo: Prentice Hall, 2006.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos.** 2ª ed. Rio de Janeiro: LTC, 1994.

TANENBAUM, A. M., LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de Dados Usando C.** São Paulo: Makron Books, 1995.

VELOSO, P. E. **Estruturas de Dados.** 4ª ed. Rio de Janeiro: Campus, 1986.

VILLAS, M. E. **Estruturas de Dados: conceitos e técnicas de implementação.** Rio de Janeiro: Campus, 1993.

3

Uso das Estruturas de Dados – Lista Linear Sequencial

No capítulo 1 estudamos as principais características das listas lineares. Vimos que as listas lineares são estruturas de dados que têm como objetivo armazenar um conjunto de dados, que de alguma forma se relacionam, com os elementos dispostos em sequência.

As listas lineares podem ser representadas quanto a sua forma de armazenamento, de duas formas: sequencial (contígua) ou encadeada. A forma contígua é considerada a maneira mais simples de armazenar uma estrutura de lista na memória. Isto porque os elementos da lista ocupam posições consecutivas na memória do computador. Esta abordagem traz como vantagem o acesso a qualquer elemento da lista de forma direta e tempo constante.

Neste sentido, iremos nos aprofundar nas próximas seções, sobre as listas lineares sequenciais (contígua) e suas variações como pilhas e filas. Adicionalmente, estudaremos diversas formas de ordenação de dados.

OBJETIVOS

- Aprofundaremos os nossos estudos teóricos e práticos a respeito das estruturas de dados;
- Compreender as variações e operações dos tipos de estruturas.

3.1 Conceito das Estruturas de Dados – Lista Linear

As listas lineares podem ser representadas fisicamente, ou quanto a sua forma de armazenamento, de duas formas: sequencial (contígua) ou encadeada.

Em listas lineares sequenciais, os dados são armazenados em endereços de memória sequencial ou contíguos, ou seja, um dado após o outro. Dessa forma temos uma estrutura estática. Em estrutura estática, a alocação de memória é feita durante a compilação do programa. Consequentemente, deve-se pré-determinar a quantidade máxima de elementos da lista.

As listas lineares sequenciais são ideais para conjunto pequeno de dados. Os dados são armazenados nos nós da lista em formato de campos, isto quando a lista linear possui dado composto (figura 3.1). Cada nó da lista, geralmente possui um identificador distinto chamado chave. No exemplo abaixo o campo chave é o Cód Cliente.

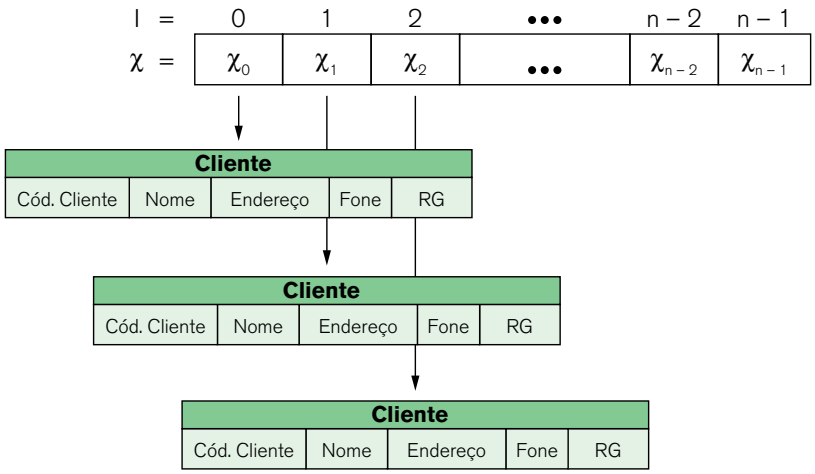


Figura 3.1 – Exemplo de nó da lista.

Pode-se ordenar ou não a lista de acordo com os campos chaves. São denominadas listas ordenadas no primeiro caso e listas não ordenadas no segundo. Para (SZWARCFITER & MARKENZON, 1994) o armazenamento sequencial é altamente eficiente quando utilizados em filas e pilhas. Isto devido a eficiência de implementação das operações básicas utilizando armazenamento

sequencial para estes tipos de estruturas. No entanto, Szwarcfiter e Markenzon (1994) alertam que um estudo aprofundado deve ser feito quando se empregam diversas estruturas simultaneamente. Devido ao fato de diversas estruturas serem onerosas no uso de memória. Conclui-se, dessa forma, que o uso da lista linear sequencial é indicado em aplicações onde há necessidade de poucos elementos e que se possa estimar o tamanho máximo de elementos da lista.

Vantagens da lista linear sequencial:

- Qualquer elemento da lista pode ser acessado direto indexado.
- Tempo constante para acesso a qualquer elemento da lista.

Desvantagem da lista linear sequencial:

- Movimentação de todos os elementos da lista quando há uma inserção ou remoção de elemento.
- O tamanho máximo da lista deve ser pré-estimado.

3.1.1 Diferenças entre Lista Linear Sequencial e Encadeada.

Uma das vantagens das listas lineares encadeadas sobre as sequenciais é que não há a necessidade de pré-determinar a quantidade máxima de elementos da lista. Os elementos não são armazenados de forma contígua e sim, ocupam qualquer posição de memória disponível.

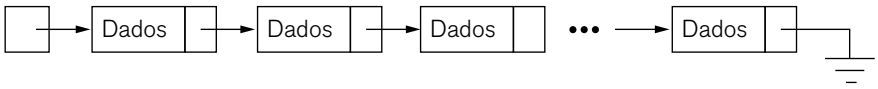


Figura 3.2 – Lista linear encadeada

A ligação entre os elementos dispersos na memória são feitos através de ponteiros. Ou seja, cada nó, deve conter, além dos registros (dados propriamente ditos), uma referência para o próximo nó da lista (figura 3.2). Assim a quantidade máxima de elementos é determinada pela quantidade de memória disponível que pode ser alocada pelo programa. Dessa forma temos uma estrutura dinâmica. Em estruturas dinâmicas a alocação de memória é feita durante a execução do programa.

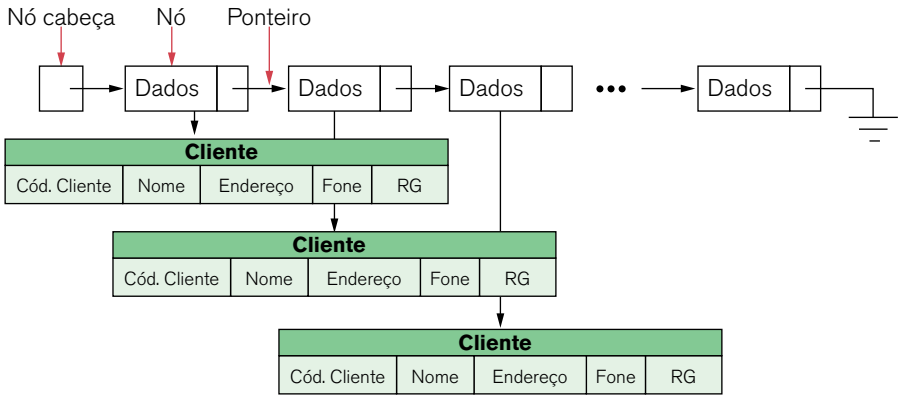


Figura 3.3 – Lista linear encadeada.

A lista possui um nó especial, chamado nó-cabeça. O nó-cabeça indica o início da lista encadeada e nunca pode ser removido. Dados (registros) do tipo que são armazenados nos demais nós da lista, não devem ser armazenados no nó-cabeça.

> Segundo Szwarcfiter e Markenzon (1994), alguns dados relativos a implementação da lista podem ser armazenados no nó-cabeça.

Varias são as operações feitas pelas listas lineares encadeadas. Para o melhor entendimento, será apresentada as operações de inserção e remoção de elemento da lista. Parte se do princípio de que a lista esta ordenada pelo campo chave. Assim temos:

- Inserir um elemento na lista:

A partir do no-cabeça, efetua-se a busca da posição onde o novo nó deve ser inserido (figura 3.4a).

Uma vez encontrado o local de inserção, liga o ponteiro nó anterior ao novo nó (figura 3.4b).

Liga o ponteiro no novo nó ao nó posterior (figura 3.4b).

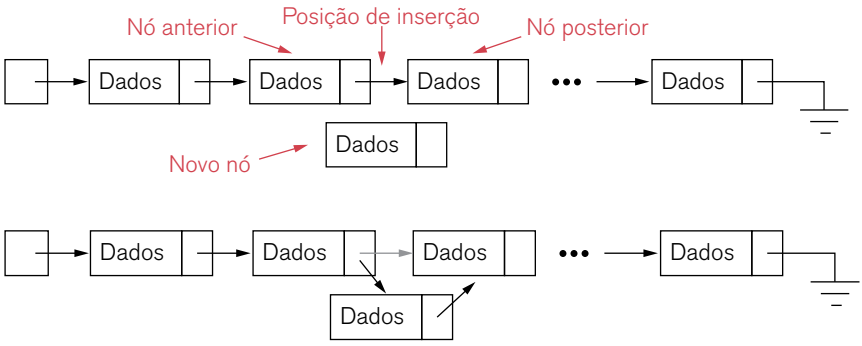


Figura 3.4 – Inserção de um nó.

• Remover um elemento na lista:

A partir do no-cabeça, efetua-se a busca do nó a ser removido (Figura 23).

Uma vez encontrado nó a ser removido, liga o ponteiro do nó anterior ao nó posterior (figura 3.5).

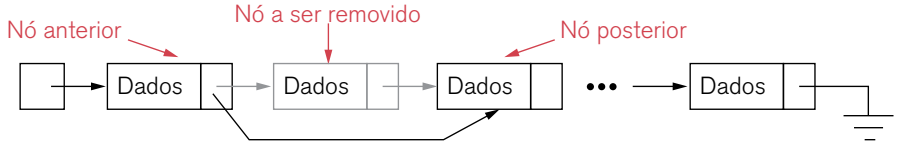


Figura 3.5 – Remoção de um nó.

Vantagens da lista linear encadeada:

• Facilidade de inserir ou remover um elemento em qualquer ponto da lista.

• Não há a necessidade de movimentar os elementos da lista quando há uma inserção ou remoção de elemento.

Desvantagem da lista linear encadeada:

• Por utilizar ponteiros, a implementação deve ser feita com muito cuidado para que não ocorra um mal encadeamento (ligação) dos nós e consequentemente a lista seja perdida.

• Encontrar um determinado elemento na posição n da lista é necessário percorrer os n-1 anteriores.

• Necessidade de memória extra para armazenamento dos ponteiros.

3.2 Uso das estruturas de dados – Lista linear sequencial.

As listas lineares sequenciais são estruturas estáticas que têm como objetivo representar um conjunto de dados, que de alguma forma se relacionam, com os elementos dispostos em sequência (figura 3.6).

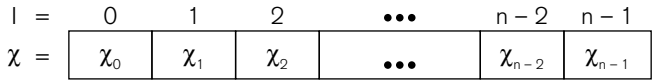


Figura 3.6 - Listas Lineares Sequenciais

Pelo fato das listas lineares sequenciais serem estrutura estática, a alocação de memória é feita durante a compilação do programa. Consequentemente, deve-se pré-determinar a quantidade máxima de elementos da lista. Os elementos armazenados nos nós podem conter dado primitivo (inteiro, `string` etc.) (figura 3.7) ou dado composto (registro – figura 3.8).

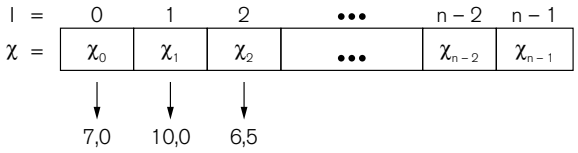


Figura 3.7 - Dado primitivo (Nota de alunos).

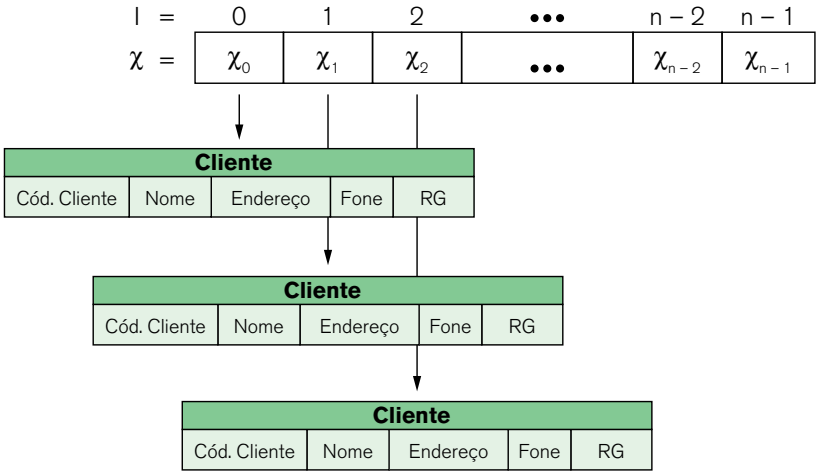


Figura 3.8 – Dado composto (Cadastro de cliente).

3.3 Principais características da Lista Linear Sequencial

Uma lista linear L é uma coleção L:[x_1, x_2, ..., x_n], n>=0 tal que:

- x_1 é o primeiro elemento da lista.
- x_n é o último elemento da lista.
- x_k, x<k<n, é seguido do elemento x_{k+1} e precedido de x_{k-1} na lista.

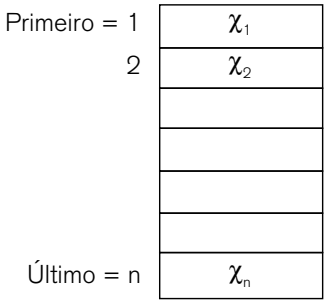


Figura 3.9 – Implementação de uma lista.. Fonte: Ziviani (1996).

3.4 Operações básicas com listas sequenciais

As operações implementadas em uma lista dependem do tipo de aplicação. Algumas operações mais comuns são:

- **Criar**: cria uma lista vazia.
- **Verificar lista vazia**: verifica se há algum elemento na lista.
- **Verificar lista cheia**: verifica se a lista esta cheia.
- **Inserir**: insere um elemento numa determinada posição ou no final da lista.
- **Alterar**: alterar algum elemento da lista.
- **Remover**: remove um elemento de uma determinada posição.
- **Buscar**: acessa um elemento da lista.
- **Exibir a quantidade**: retorna a quantidade de elementos da lista.
- **Combinar**: combina duas ou mais listas em uma única.
- **Dividir lista**: dividi uma lista em duas ou mais.
- **Ordenar**: ordena os elementos da lista de acordo com algum de seus componentes.
- **Esvaziar**: esvaziar a lista.

A seguir, serão implementadas algumas das operações acima descritas. Como o nó de uma lista pode conter dados do tipo primitivo ou compostos, será apresentado a implementação para estas duas possibilidades.

> A forma como as estruturas de dados são implementadas é o segredo da velocidade de muitos programas. Isto porque, nem sempre é tão óbvia a melhor forma de organizar os dados durante o processamento. Mas as estruturas de dados facilitam este processo e influenciam diretamente o desempenho dos programas.

3.4.1 Estruturas homogêneas - Criar uma lista vazia

Para definir uma lista primitiva de elementos, como por exemplo, para guardar as notas de alunos, pode ser utilizado um vetor. A lista linear é definida pelo `array ListaNotas[...]`. A quantidade máxima de elementos da lista é informada na constante MAX_LISTA. Já PosUltimoElemLista tem três finalidades. Primeiramente é controlar a quantidade de elementos da lista para que não ultrapasse a quantidade máxima permitida. Segundo, indicar a posição de inserção de um novo elemento, caso este seja inserido no final da lista. Terceiro, indicar a posição do último elemento válido na lista. Por fim `Ret` recebe o valor de retorno, da função chamada, indicando se houve ou não sucesso na operação realizada. A figura 3.10 ilustra a lista e as variáveis de controle descritas anteriormente.

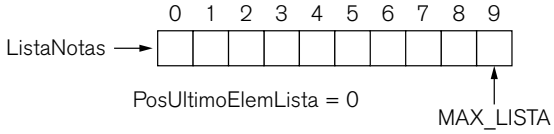


Figura 3.10 – Lista linear sequencial vazia

```
#include <iostream>
#include "conio.h"
using namespace std;

# define MAX_LISTA 10      // Tamanho Máximo da Lista
```

```
void main(){

    float ListaNotas[MAX_LISTA]; // Lista Linear Sequencial
    int PosUltimoElemLista = 0;  // Qtde de elementos da lista
    bool Ret; // Recebe o retorno da função chamada
}
```

3.4.2 Estruturas homogêneas - Inserir um elemento na lista

A função Inserir(...) insere um novo elemento no final da lista. Para tanto, recebe o vetor Lista[] por referência, o valor a ser inserido e a variável de controle de elementos da lista PosUltimoElem. Primeiramente a função verifica se a lista possui espaço para inserir o novo elemento. Se a lista não estiver cheia, o elemento é inserido no final da lista e a variável de controle PosUltimoElem é incrementada em uma unidade. Caso a lista esteja cheia, é dado um aviso para o usuário e a função retorna falso, indicando que a operação não foi realizada.

```
bool Inserir(float Lista[], float valor, int &PosUltimoElem) {

      if (PosUltimoElem == MAX_LISTA){
            cout << "ERRO: Lista cheia.";
            return false;
      }
      else {
            Lista[PosUltimoElem] = valor;
            PosUltimoElem++;
      }
            return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Digite o valor para inserção: ";
cin >> valor;
Ret = Inserir(ListaNotas, valor, PosUltimoElemLista);
if(Ret == true){ cout << "Inserção efetuada com sucesso!" << endl;
}
```

3.4.3 Estruturas homogêneas - Inserir um elemento em uma posição determinada

Para inserir um elemento X na posição p ($1 \leq p \leq n+1$), os elementos x_i devem ser deslocados para a posição x_(i+1), ($p \leq i \leq n$). Em seguida o elemento é inserido na posição definida e é acrescido em uma unidade o total de elementos da lista.

```
bool InserirPos(float Lista[], int PosIns, float valor,
                int &PosUltimoElem) {
    int ind;
    if (PosUltimoElem == MAX_LISTA){
           cout << "ERRO: Lista cheia.";
           return false;
    }
    else {
           for(ind = PosUltimoElem + 1; ind > PosIns; ind-- ){
           Lista[ind] = Lista[ind-1];
           }
           Lista[PosIns] = valor;
           PosUltimoElem++;
}
           return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Digite o valor para inserção: ";
cin >> valor;
cout << "Digite o posição para inserção: ";
cin >> pos;
Ret = InserirPos(ListaNotas, pos, valor, PosUltimoElemLista);
if(Ret == true){ cout << "Inserção efetuada com sucesso!" << endl;}
```

3.4.4 Estruturas homogêneas - Exibir toda a lista

A função a seguir exibe todos os elementos inseridos na lista. Note que, independente do tamanho máximo definido para a lista em MAX_LISTA, serão exibidos apenas os elementos inseridos até o momento, controlados por PosUltimoElemLista.

```
bool Exibir(float Lista[], int PosUltimoElem){
    int ind;

    if (PosUltimoElem == 0){
           cout << "ERRO: Lista vazia.";
           return false;
    }

    for(ind = 0; ind < PosUltimoElem; ind++ ){
           cout << "Nota " << ind << ": " <<  Lista[ind] <<endl; }
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
Ret = Exibir(ListaNotas, PosUltimoElemLista);
if(Ret == false){ cout << "Não foi possível exibir a lista." <<
endl;}
```

3.4.5 Estruturas homogêneas - Pesquisar um determinado elemento e retornar sua posição

A função Pesquisar(...) varre os elementos da lista buscando o valor informado. Se encontrar, retorna a posição do elemento, caso contrário, retorna -1 para indicar que o elemento não faz parte da lista.

```
int Pesquisar(float Lista[], float valor, int &PosUltimoElem) {
    int ind;
    for(ind = 0; ind < PosUltimoElem; ind++ ){
            if (Lista[ind] == valor){
                    return ind;
            }
    }
    return -1;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Digite o valor da pesquisa: ";
cin >> valor;
pos = Pesquisar(ListaNotas, valor, PosUltimoElemLista);
cout << "O valor foi encontrado na posição: " << pos;
```

3.4.6 Estruturas homogêneas - Remover um elemento em uma posição determinada

Para remover um elemento X na posição p ($1 \leq p \leq n+1$), os elementos x_i devem ser deslocados para a posição x_(i-1), ($p \leq i \leq n$). Em seguida é decrescido em uma unidade o total de elementos da lista.

```
bool RemoverElem(float Lista[], int valor, int &PosUltimoElem) {
    int ind;
    int PosRem;

    if (PosUltimoElem == 0){
            cout << "ERRO: Lista vazia.";
            return false;
}
PosRem = Pesquisar(Lista, valor, PosUltimoElem);
for(ind = PosRem; ind < PosUltimoElem; ind++ ){
    Lista[ind] = Lista[ind+1];
}
```

```
PosUltimoElem--;
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Digite o valor para remoção: ";
cin >> valor;
Ret = RemoverElem(ListaNotas, valor, PosUltimoElemLista);
if(Ret == true){ cout << "Remoção efetuada com sucesso!" << endl; }
```

3.4.7 Estruturas heterogêneas - Criar uma lista vazia

Os dados (registro) são presentados pela estrutura DADOS_ALUNO. A lista linear é definida pelo `array ListaDeAlunos` [...]. As demais variáveis são as mesmas utilizadas anteriormente.

```
#include <iostream>
#include "conio.h"
using namespace std;

# define MAX_LISTA 5 // Tamanho Máximo da Lista Linear Sequencial

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};

void main(){

    DADOS_ALUNO ListaDeAlunos[MAX_LISTA]; // Lista Linear
    int PosUltimoElemLista = 0; // Qtde de elementos da lista
    bool Ret; // Recebe o retorno da função chamada
}
```

3.4.8 Estruturas heterogêneas - Inserir um elemento na lista

A função Inserir(...) insere um novo elemento no final da lista. Os dados (registros) são passados individualmente para a função e atribuídos a estrutura.

```
bool Inserir(DADOS_ALUNO Lista[], int CodAluno, char Nome[],
            int Turma, int &PosUltimoElem) {

    if (PosUltimoElem == MAX_LISTA){
            cout << "ERRO: Lista cheia.";
            return false;
    }
    else {
            Lista[PosUltimoElem].CodAluno = CodAluno;
            strcpy(Lista[PosUltimoElem].Nome, Nome);
            Lista[PosUltimoElem].Turma = Turma;

            PosUltimoElem++;
    }
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Inserção: " << endl;
cout << "Digite o código do aluno: ";
cin >> CodAluno;
cout << "Digite o nome do aluno: ";
cin >> Nome;
cout << "Digite a turma: ";
cin >> Turma;

Ret = Inserir(ListaDeAlunos, CodAluno, Nome, Turma,
PosUltimoElemLista);
if(Ret == true){cout << "Inserção efetuada com sucesso!" << endl;}
```

3.4.9 Estruturas heterogêneas - Inserir um elemento em uma posição determinada

Para inserir um elemento X na posição p ($1 \leq p \leq n+1$), os elementos x_i devem ser deslocados para a posição x_(i+1), ($p \leq i \leq n$). Em seguida o elemento é inserido na posição definida e é acrescido em uma unidade o total de elementos da lista.

```
bool InserirPos(DADOS_ALUNO Lista[], int PosIns, int CodAluno,
                  char Nome[], int Turma, int &PosUltimoElem) {
    int ind;

    if (PosUltimoElem == MAX_LISTA){
           cout << "ERRO: Lista cheia.";
           return false;
    }
    else {

           // Desloca os itens da lista
           for(ind = PosUltimoElem + 1; ind > PosIns; ind-- ){
                  Lista[ind] = Lista[ind-1];
           }
           // Insere novo elemento
           Lista[PosIns].CodAluno = CodAluno;
           strcpy(Lista[PosIns].Nome, Nome);
           Lista[PosIns].Turma = Turma;
           PosUltimoElem++;
    }
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Inserção: " << endl;
cout << "Digite o código do aluno: ";
cin >> CodAluno;
```

```
cout << "Digite o nome do aluno: ";
cin >> Nome;
cout << "Digite a turma: ";
cin >> Turma;
cout << "Digite o posição para inserção: ";
cin >> pos;
Ret = InserirPos(ListaDeAlunos,  pos, CodAluno, Nome, Turma,
 PosUltimoElemLista);
if(Ret == true){cout << "Inserção efetuada com sucesso!" << endl;}
```

3.4.10 Estruturas heterogêneas - Exibir toda a lista

A função a seguir exibe todos os elementos inseridos na lista. Note que, independente do tamanho máximo definido para a lista em MAX_LISTA, serão exibidos apenas os elementos inseridos até o momento, controlados por PosUltimoElemLista.

```
bool Exibir(DADOS_ALUNO Lista[], int PosUltimoElem){
  int ind;
  if (PosUltimoElem == 0){
            cout << "ERRO: Lista vazia.";
            return false;
  }
  for(ind = 0; ind < PosUltimoElem; ind++ ){
    cout << "Código do Aluno: " << Lista[ind].CodAluno <<endl;
    cout << "Nome: " << Lista[ind].Nome <<endl;
    cout << "Turma: " <<  Lista[ind].Turma <<endl;
}
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
Ret = Exibir(ListaDeAlunos, PosUltimoElemLista);
if(Ret == false){cout << "Não foi possível exibir a lista." << endl;}
```

3.4.11 Estruturas heterogêneas - Pesquisar um determinado elemento e retornar sua posição

A função Pesquisar(...) varre os elementos da lista buscando o código do aluno informado. Se encontrar, retorna a posição do elemento, caso contrário, retorna –1 para indicar que o elemento não faz parte da lista.

```
int Pesquisar(DADOS_ALUNO Lista[], int CodAluno, int &PosUltimoElem) {
    int ind;
    for(ind = 0; ind < PosUltimoElem; ind++ ){
           if (Lista[ind].CodAluno == CodAluno){
                  return ind;
           }
    }
    return -1;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Digite o código do aluno: ";
cin >> CodAluno;
pos = Pesquisar(ListaDeAlunos, CodAluno, PosUltimoElemLista);
cout << "O código do aluno foi encontrado na posição: " << pos;
```

3.4.12 Estruturas heterogêneas - Remover um elemento em uma posição determinada

Para remover um elemento X na posição p ($1 \leq p \leq n+1$), os elementos x_i devem ser deslocados para a posição x_(i-1), ($p \leq i \leq n$). Em seguida é decrescido em uma unidade o total de elementos da lista.

```
bool RemoverElem(DADOS_ALUNO Lista[], int CodAluno,
                 int &PosUltimoElem) {
    int ind;
    int PosRem;

    if (PosUltimoElem == 0){
           cout << "ERRO: Lista vazia.";
           return false;
    }
    // Encontra a posição do elemento que será removido
    PosRem = Pesquisar(Lista, CodAluno, PosUltimoElem);

    // Desloca a lista sobrepondo o elemento a ser removido
    for(ind = PosRem; ind < PosUltimoElem; ind++ ){
           Lista[ind] = Lista[ind+1];
    }

    PosUltimoElem--;
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Digite o código do aluno para remoção: ";
cin >> CodAluno;
Ret = RemoverElem(ListaDeAlunos, CodAluno, PosUltimoElemLista);
if(Ret == true){cout << "Remoção efetuada com sucesso!" << endl;}
```

CONEXÃO

Para entender melhor a implementação de listas sequenciais, veja o link: http://www.ft.unicamp.br/liag/siteEd/implementacao/lista-ligada-estruturada-estatica.php. O autor implementa diversas operações com listas sequenciais estáticas.

3.5 Aplicação dos conceitos de ordenação e pesquisa com Lista Linear Sequencial

Pelas operações descritas anteriormente, percebe se a quantidade de aplicações que podem utilizar as listas sequenciais. O próprio exemplo apresentado, a respeito da manutenção dos dados de alunos, é uma aplicação com algumas das principais operações feitas em programas de cadastramento. Pode se estender estes conceitos para aplicações do tipo: cadastro de produtos, em um sistema de estoque, cadastro de veículos e clientes em um sistema de concessionária de veículos, sistemas bibliotecários, escolares, etc.

Todos estes programas utilizam de alguma forma conceitos de ordenação e pesquisa com Lista Linear Sequencial. No capítulo 2 implementamos diversos algoritmos de ordenação e pesquisa com Lista Linear Sequencial. Caso você tenha alguma dúvida, seria importante revisar o conteúdo do capítulo 2.

3.6 Estrutura de Dados Pilha

A pilha é um tipo especial de lista onde os elementos a serem inseridos ou removidos ocorrem no topo da pilha. Estruturas do tipo pilha são as mais utilizadas em programação por ser uma das estruturas de dados mais simples. Muitas máquinas modernas tem implementado pilhas diretamente no hardware. Sua característica de empilhar dados, por exemplo, são uteis em chamadas de funções. Na chamada de funções, as variáveis locais são empilhadas na pilha. Quando a função é finalizada, as variáveis locais são desempilhadas.

3.7 Representação da Estrutura de Dados Pilha por CONTIGUIDADE

Como foi descrito no capítulo 1, a pilha é um tipo especial de lista onde os elementos a serem inseridos ou removidos ocorrerão no topo da pilha (Figura 29). Esta característica é conhecida como LIFO (Last In, First Out - Último a Entrar,

Primeiro a Sair). Dessa forma, o último elemento que foi inserido na pilha será o primeiro elemento a ser removido.

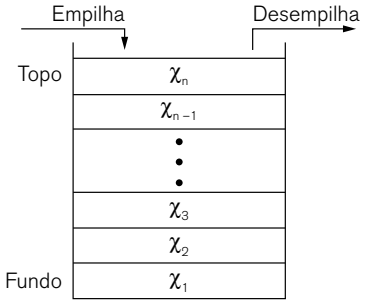


Figura 3.11 – Pilha.

Uma pilha P é uma coleção P:[x_1, x_2, ..., x_n], n>=0 tal que:

- x_1 é o primeiro elemento da lista.
- x_n é o último elemento da lista.
- x_k, x<k<n, é seguido do elemento x_{k+1} e precedido de x_{k-1} na pilha.

3.8 Operações básicas com pilha

Algumas operações mais comuns em pilha são:

- **Criar**: cria uma pilha vazia.
- **Empilhar**: insere um novo elemento no topo da pilha.
- **Desempilhar**: remove um elemento do topo da pilha.
- **Exibir topo**: exibe o elemento do topo da pilha.
- **Exibir a quantidade**: retorna a quantidade de elementos da pilha.
- **Esvaziar**: esvazia todos os elementos da pilha.

A seguir, serão implementadas algumas das operações acima descritas. As operações serão feitas com dados compostos.

3.9 Teste de aplicação com Pilha sequencial.

3.9.1 Criar uma pilha vazia

A pilha é definida pelo `array PilhaAlunos` [...]. A quantidade máxima de elementos da lista é informada na constante MAX_PILHA. Já PosTopo tem duas finalidades. Primeiramente é controlar a quantidade de elementos da pilha para que não ultrapasse a quantidade máxima permitida. Segundo, indicar a posição do topo para a inserção de um novo elemento. Por fim `Ret` recebe o valor de retorno, da função chamada, indicando se houve ou não sucesso na operação realizada. A figura 3.12 ilustra a lista e as variáveis de controle descritas anteriormente.

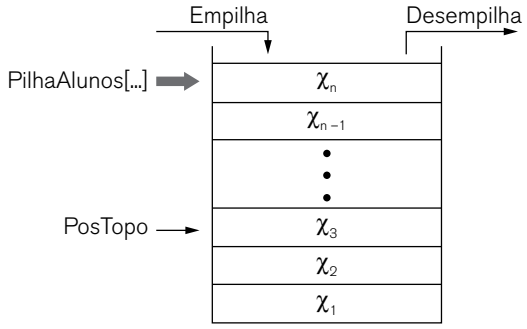


Figura 3.12 – Pilha.

```
#include <iostream>
#include "conio.h"
using namespace std;

# define MAX_PILHA 5       // Tamanho Máximo da Pilha

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};
```

```
void main(){

    DADOS_ALUNO PilhaAlunos[MAX_PILHA];// Pilha
    int PosTopo = 0;      // Posição do topo
    bool Ret;             // Recebe o retorno da função chamada

    system("pause>null");
}
```

3.9.2 Empilhar (Push)

A função Empilhar(...) insere um novo elemento no topo da pilha. Para tanto, recebe o vetor Pilha[] por referência, o valor a ser inserido e a variável de controle de elementos da lista PosTopo. Primeiramente a função verifica se a pilha possui espaço para inserir o novo elemento. Se a lista não estiver cheia, o elemento é inserido. Caso contrário, é dado um aviso para o usuário e a função retorna falso, indicando que a operação não foi realizada.

```
bool Empilhar(DADOS_ALUNO Pilha[], int CodAluno, char Nome[],
                  int Turma, int &PosTopo) {

    if (PosTopo == MAX_PILHA){
           cout << "ERRO: Pilha cheia.";
           return false;
    }
    else {
           Pilha[PosTopo].CodAluno = CodAluno;
           strcpy(Pilha[PosTopo].Nome, Nome);
           Pilha[PosTopo].Turma = Turma;

           PosTopo++;
    }
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Inserção: " << endl;
cout << "Digite o código do aluno: ";
cin >> CodAluno;
cout << "Digite o nome do aluno: ";
cin >> Nome;
cout << "Digite a turma: ";
cin >> Turma;

Ret = Empilhar(PilhaAlunos, CodAluno, Nome, Turma, PosTopo);
if(Ret == true){cout << "Inserção efetuada com sucesso!" << endl;}
```

3.9.3 Exibir o topo da pilha (Stacktop)

A função ExibirTopo(...) tem como objetivo exibir o elemento do topo da pilha.

```
bool ExibirTopo(DADOS_ALUNO Pilha[], int PosTopo){

    if (PosTopo == 0){
           cout << "ERRO: Pilha vazia.";
           return false;
    }

    // Exibe o elemento do TOPO da pilha
    cout << "Código do Aluno: " << Pilha[PosTopo - 1].CodAluno
<<endl;
    cout << "Nome: " << Pilha[PosTopo - 1].Nome <<endl;
    cout << "Turma: " <<  Pilha[PosTopo - 1].Turma <<endl;
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
Ret = ExibirTopo(PilhaAlunos, PosTopo);
if(Ret == false){cout << "Não foi possível exibir a pilha." << endl;}
```

3.9.4 Exibir toda a pilha (Pop)

A função a seguir exibe todos os elementos inseridos na pilha. A exibição dos dados é feita do último elemento inserido para o primeiro, ou seja, exibe do topo para o fundo da pilha. Note que, independente do tamanho máximo definido para a lista em MAX_PILHA, serão exibidos apenas os elementos inseridos até o momento, controlados por PosTopo.

```
bool Exibir(DADOS_ALUNO Pilha[], int PosTopo){
    int ind;

    if (PosTopo == 0){
                cout << "ERRO: Pilha vazia.";
                return false;
    }
    // Exibe do TOPO para o FUNDO
    for(ind = PosTopo - 1; ind >= 0; ind-- ){
           cout << "Código do Aluno: " << Pilha[ind].CodAluno
<<endl;
           cout << "Nome: " << Pilha[ind].Nome <<endl;
           cout << "Turma: " <<  Pilha[ind].Turma <<endl;
  }
  return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
Ret = Exibir(PilhaAlunos, PosTopo);
if(Ret == false){cout << "Não foi possível exibir a pilha." <<
endl;}
```

3.9.5 Desempilhar

A função Desempilhar(...) remove o elemento no topo da pilha, caso esta não esteja vazia. Note que para desempilhar, basta decrementar a variável PosTopo. Isto porque o controle para inserir um novo elemento no topo da pilha, como também, a exibição dos elementos da pilha ou do elemento do topo, é controlado por esta variável.

```
bool Desempilhar(int &PosTopo) {

        if (PosTopo == 0){
                cout << "ERRO: pilha vazia.";
                return false;
        }
        else {
                // Desempilha o elemento do topo
                PosTopo--;
        }
        return true;
    }
```

Abaixo segue um exemplo de chamada da função.

```
Ret = Desempilhar(PosTopo);
if(Ret == false){cout << "Não foi possível desempilhar a pilha."
<< endl;}
```

CONEXÃO

Para entender melhor a implementação de pilhas, veja o link: http://www.devmedia.com.br/pilhas-fundamentos-e-implementacao-da-estrutura-em-java/28241

3.10 Estrutura de dados - Fila simples e Fila Circular.

A fila é um conceito que é conhecido naturalmente. A fila esta presente em bancos, entrada de cinema, caixa de supermercado etc. De mesma forma que a pilha, a fila é um tipo partícula de lista linear. A fila é a estrutura ideal para ser utilizada em aplicações em que, no armazenamento de dados, é necessário preservar a ordem do primeiro que entrar é o primeiro a sair.

3.11 Representação da estrutura de dados Fila por contiguidade (Fila simples).

O acesso aos dados da fila é feito através de FIFO (`First In`, `First Out` - Primeiro a Entrar, Primeiro a Sair) (figura 3.13), ou seja, os elementos são inseridos em uma extremidade, e retirados na extremidade oposta.

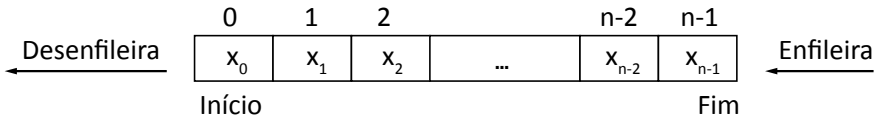


Figura 3.13 - Fila.

Uma fila P é uma coleção F:[x_1, x_2, ..., x_n], n>=0 tal que:

- x_1 é o primeiro elemento da lista.
- x_n é o último elemento da lista.
- x_k, x<k<n, é seguido do elemento x_{k+1} e precedido de x_{k-1} na fila.

3.12 Operações com Fila simples.

As operações mais comuns efetuadas com filas são:

- Criar: cria uma fila vazia.
- Enfileirar: insere um elemento no fim da fila.
- Desenfileirar: remover um elemento no início da fila.
- Exibir início: exibe o elemento do início da fila.
- Exibir a quantidade: retorna a quantidade de elementos da fila.
- Esvaziar: esvazia a fila.

A seguir, serão implementadas algumas das operações acima descritas. As operações serão feitas com dados compostos.

3.12.1 Criar uma fila vazia

A fila é definida pelo `array FilaAlunos[...]`. A quantidade máxima de elementos da fila é informada na constante MAX_FILA. A variável IniFila é utilizada para obter o primeiro elemento da fila. A variável FimFila controla a quantidade de elementos da pilha para que não ultrapasse a quantidade máxima permitida e indica a posição para a inserção de um novo elemento. Por fim `Ret` recebe o valor de retorno, da função chamada, indicando se houve ou não sucesso na operação realizada. A figura 3.14 ilustra a fila e as variáveis de controle descritas anteriormente.

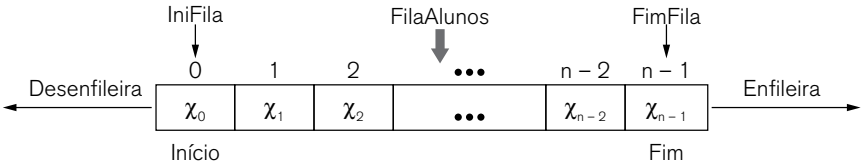


Figura 3.13 – Fila.

```
#include <iostream>
#include "conio.h"
using namespace std;

# define MAX_FILA 5        // Tamanho Máximo da Fila

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};

void main(){

    DADOS_ALUNO FilaAlunos[MAX_FILA];   // Fila
    int IniFila= 0;                     // Inicio da fila
    int FimFila = 0;                    // Fim da fila
    bool Ret;             // Recebe o retorno da função chamada
    system("pause>null");
}
```

A operação enfileiramento incrementa a variável FimFila a medida que um novo elemento é inserido na fila. O desenfileiramento apenas incrementa a variável IniFila sem realmente remover o elemento fisicamente do vetor.

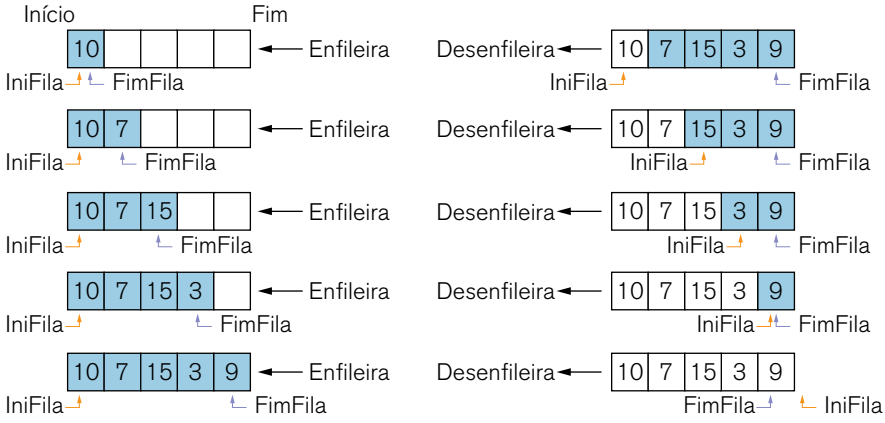


Figura 3.15 – Enfileiramento e Desenfileiramento de elementos na fila.

3.12.2 Enfileirar um elemento

A função Enfileirar(...) insere um novo elemento na fila.

```
   bool Enfileirar(DADOS_ALUNO Fila[], int CodAluno, char Nome[], int
Turma, int &FimFila) {

        if (FimFila == MAX_FILA){
                cout << "ERRO: Fila cheia.";
                return false;
        }
        else {
                Fila[FimFila].CodAluno = CodAluno;
                strcpy(Fila[FimFila].Nome, Nome);
                Fila[FimFila].Turma = Turma;

                FimFila++;
        }
        return true;
   }
```

Abaixo segue um exemplo de chamada da função.

```
cout << "Inserção: " << endl;
cout << "Digite o código do aluno: ";
cin >> CodAluno;
cout << "Digite o nome do aluno: ";
cin >> Nome;
cout << "Digite a turma: ";
cin >> Turma;

Ret = Enfileirar(FilaAlunos, CodAluno, Nome, Turma, FimFila);
if(Ret == true){cout << "Inserção efetuada com sucesso!" << endl;}
```

3.12.3 Exibir o primeiro elemento da fila

A função ExibirPrimeiro() exibe o primeiro elemento da fila.

```
bool ExibirPrimeiro(DADOS_ALUNO Fila[], int IniFila, int FimFila){
    if (FimFila == 0){
            cout << "ERRO: Fila vazia.";
            return false;
    }

    // Exibe o primeiro elemento da fila
    cout << "Código do Aluno: " << Fila[IniFila].CodAluno <<endl;
    cout << "Nome: " << Fila[IniFila].Nome <<endl;
    cout << "Turma: " <<  Fila[IniFila].Turma <<endl;

    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
Ret = ExibirPrimeiro(FilaAlunos,  IniFila, FimFila);
if(Ret == false){cout << "Não foi possível exibir a fila." << endl;}
```

3.12.4 Desenfileirar um elemento

A função Desenfileirar(...) remove o elemento do início da fila. Para isso, apenas incrementa a variável IniFila sem realmente remover o elemento fisicamente do vetor.

```
bool Desenfileirar(DADOS_ALUNO Fila[], int &PosTopo) {
    int ind;
    int PosRem;
    if (PosTopo == 0){
            cout << "ERRO: Lista vazia.";
            return false;
    }

    // Desloca a pilha sobrepondo o primeiro elemento
    for(ind = 0; ind < PosTopo; ind++ ){
            Fila[ind] = Fila[ind+1];
    }
    PosTopo--;
    return true;
}
```

Abaixo segue um exemplo de chamada da função.

```
Ret = Desenfileirar(FilaAlunos, IniFila, FimFila);
if(Ret == false){cout << "Não foi possível desenfileirar a fila." <<
endl;}
```

3.13 Estrutura de dados Fila por contiguidade (Fila circular).

A fila circular considera um vetor de forma que sua posição final se liga à posição inicial (figura 3.16). Apesar de tratar a organização física dos dados diferente da fila simples, a fila circular ainda é uma fila. Ou seja, o primeiro elemento

a entrar é o primeiro a sair da fila.

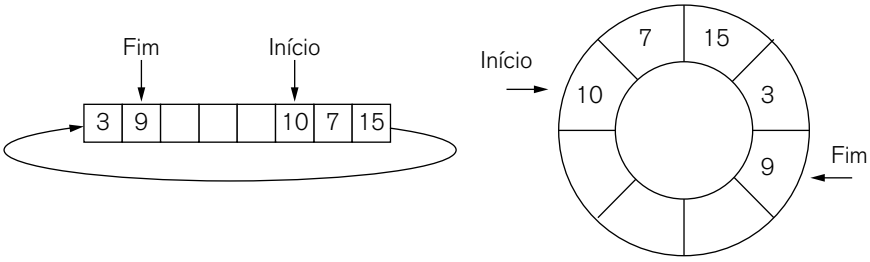


Figura 3.16 – Fila circular.

3.14 Operações com Fila circular.

As operações mais comuns efetuadas com filas são:

- **Criar**: cria uma fila vazia.
- **Enfileirar**: insere um elemento no fim da fila.
- **Desenfileirar**: remover um elemento no início da fila.
- **Exibir início**: exibe o elemento do início da fila.
- **Exibir a quantidade**: retorna a quantidade de elementos da fila.
- **Esvaziar**: esvazia a fila.

A seguir, serão implementadas algumas das operações acima descritas. As operações serão feitas com dados compostos.

3.14.1 Criar uma fila vazia

A fila é definida pelo `array FilaAlunos`[...]. A quantidade máxima de elementos da fila é informada na constante MAX_FILA. A variável IniFila é utilizada para obter o primeiro elemento da fila. A variável FimFila controla a posição de inserção de um novo elemento. TotalFila controla a quantidade de elementos da pilha para que não ultrapasse a quantidade máxima permitida. Note a variável Removido na estrutura DADOS_ALUNO. Esta variável indica se o elemento foi removido ou não da lista. Caso tenha sido marcado como removido, o elemento

não será mostrado na função Exibir(...) mesmo estando fisicamente no vetor. Por fim Ret recebe o valor de retorno, da função chamada, indicando se houve ou não sucesso na operação realizada.

```
#include <iostream>
#include "conio.h"
using namespace std;

# define MAX_FILA 5 // Tamanho Máximo da Fila

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
    bool Removido; // Indica se o elemento foi removido
};

    void main(){
           DADOS_ALUNO FilaAlunos[MAX_FILA]; // Fila
           int IniFila = 0;  // Inicio da fila
           int TotalFila = 0;  // Totla de elementos da fila
           int FimFila = 0;  // Fim da fila
           bool Ret; // Recebe o retorno da função chamada

           system("pause>null");
    }
```

3.14.2 Enfileirar um elemento

A função Enfileirar(...) insere um novo elemento na fila. Note o trecho do código onde se efetua o seguinte teste `if (FimFila == MAX_FILA)`. Este teste, se for verdadeiro, indica que o elemento atual foi inserido no final da fila. Consequentemente o próximo elemento será inserido no início do vetor, caso a fila não esteja cheia.

```
   bool Enfileirar(DADOS_ALUNO Fila[], int CodAluno, char Nome[], int
Turma, int &FimFila, int &TotalFila) {

        if (TotalFila == MAX_FILA){
                cout << "ERRO: Fila cheia.";
                return false;
        }
        else {
                Fila[FimFila].CodAluno = CodAluno;
                strcpy(Fila[FimFila].Nome, Nome);
                Fila[FimFila].Turma = Turma;
                Fila[FimFila].Removido = false;

                FimFila++;

                // Se chegou no fim da fila, então o próximo
                // elemento a ser inserido será no inicio da fila
                if (FimFila == MAX_FILA)
                        FimFila = 0;
                TotalFila++;
        }
        return true;
   }
```

Abaixo segue um exemplo de chamada da função.

```
   cout << "Inserção: " << endl;
   cout << "Digite o código do aluno: ";
   cin >> CodAluno;
   cout << "Digite o nome do aluno: ";
   cin >> Nome;
   cout << "Digite a turma: ";
   cin >> Turma;
   Ret  =  Enfileirar(FilaAlunos,  CodAluno,  Nome,  Turma,  FimFila,
TotalFila);
   if(Ret == true){cout << "Inserção efetuada com sucesso!" << endl;}
```

3.14.3 Exibir

A função Exibir() exibe todos os elementos da fila menos os marcados como removido.

```
   bool  Exibir(DADOS_ALUNO  Filha[],  int  IniFila,  int  FimFila,  int
TotalFila){
     int ind;

     if (TotalFila == 0){
   cout << "ERRO: Pilha vazia.";
   return false;
     }

   if (IniFila < FimFila){
      for(ind = IniFila; ind < FimFila; ind++ ){
       cout << "Código do Aluno: " <<  Filha[ind].CodAluno <<endl;
       cout << "Nome: " <<  Filha[ind].Nome <<endl;
       cout << "Turma: " <<  Filha[ind].Turma <<endl;
      }
   }else{
      for(ind = IniFila; ind < MAX_FILA; ind++ ){
   if ( Filha[ind].Removido == false){
       cout << "Código do Aluno: " <<  Filha[ind].CodAluno <<endl;
       cout << "Nome: " <<  Filha[ind].Nome <<endl;
       cout << "Turma: " <<  Filha[ind].Turma <<endl;
   }
      }
            for(ind = 0; ind < FimFila; ind++ ){
   if ( Filha[ind].Removido == false){
       cout << "Código do Aluno: " <<  Filha[ind].CodAluno <<endl;
       cout << "Nome: " <<  Filha[ind].Nome <<endl;
       cout << "Turma: " <<  Filha[ind].Turma <<endl;
   }
      }
   }
   return true;
   }
```

Abaixo segue um exemplo de chamada da função.

```
   Ret = Exibir(FilaAlunos, IniFila,  FimFila, TotalFila);
   if(Ret == false){cout << "Não foi possível exibir a fila." << endl;
}
```

3.14.4 Desenfileirar um elemento

A função Desenfileirar(...) remove o elemento do início da fila. Fisicamente o elemento não é removido e sim marcado como removido. Note o trecho do código onde efetua-se o seguinte teste `if (IniFila == MAX_FILA)`. Este teste, se for verdadeiro, indica que o próximo elemento a ser removido da lista será o primeiro fisicamente do vetor.

```
   bool Desenfileirar(DADOS_ALUNO Fila[], int &IniFila, int &TotalFila)
{
   int ind;
   int PosRem;

   if (TotalFila == 0){
              cout << "ERRO: Fila vazia.";
              return false;
   }

   //Indica que o elemento do início da fila foi removido
   Fila[IniFila].Removido = true;
   IniFila++;

   // Se chegou no fim da fila, então o próximo
   // elemento a ser removido será no inicio da fila
   if (IniFila == MAX_FILA)
       IniFila = 0;

   TotalFila--;
   return true;
   }
```

Abaixo segue um exemplo de chamada da função.

```
   Ret = Desenfileirar(FilaAlunos,  IniFila, TotalFila);
   if(Ret == false){cout << "Não foi possível desenfileirar a fila." <<
endl;}
```

ATIVIDADES

01. O que são e para que servem as listas lineares sequencias?

02. Cite as principais vantagens e desvantagens da lista linear sequencial.

03. Para que serve o nó-cabeça em uma lista encadeada?

REFLEXÃO

Neste capítulo estudamos as listas lineares sequenciais. Abordamos as suas características, implementamos e analisamos suas principais operações. Da mesma forma, desenvolvemos as principais rotinas relacionadas as operações com fila e pilha.

Todos estes conhecimentos, certamente serão imprescindíveis para sua vida profissional.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de listas lineares sequenciais e demais assuntos deste capítulo, consulte a sugestão de link abaixo:

CELES, W.; RANGEL, J. L. Apostila de Estruturas de dados. Disponível em: http://www-usr.inf.ufsm.br/~juvizzotto/elc1067-2013b/estrut-dados-pucrio.pdf. Acesso em: Jan: 2015.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILAR, L. J. **Fundamentos de Programação: Algoritmos, Estruturas de Dados e Objetos.** 3ª ed. São Paulo: McGraw-Hill, 2008.

CELES, W.; RANGEL, J. L. **Apostila de Estruturas de dados.** Disponível em: http://www-usr.inf.ufsm.br/~juvizzotto/elc1067-2013b/estrut-dados-pucrio.pdf. Acesso em: Jan: 2015.

DEITEL, H. M., & DEITEL, P. J. **C++ Como Programar.** 3ª ed. Porto Alegre: Bookman, 2001.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos.** 2ª ed. Rio de Janeiro: LTC, 1994.

TANENBAUM, A. M., LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de Dados Usando C.** São Paulo: Makron Books, 1995.

VELOSO, P. E. **Estruturas de Dados.** 4ª ed. Rio de Janeiro: Campus, 1986.

VILLAS, M. E. **Estruturas de Dados: conceitos e técnicas de implementação.** Rio de Janeiro: Campus, 1993.

4

Ponteiros e Alocação Dinâmica

Um dos principais aspectos, que dão um poder a mais a linguagem C++ é a possibilidade de utilização de ponteiros e a alocação dinâmica de memória. Os ponteiros permitem um acesso direto a objetos e códigos na memória. Com a alocação dinâmica é possível manusear de forma mais eficiente grandes quantidades de dados. Dessa forma, ponteiros e alocação dinâmica permitem a criação e manipulação de estruturas de dados complexas que necessitam conter referências umas para outras, tais como, listas encadeadas, árvores binárias , etc.

Assim, estudaremos como a linguagem C++ permite a criação e como são as várias formas de utilização de ponteiros. Veremos também como é possível alocar memória para armazenar nossas estruturas de dados, e quais são os pontos importantes desta forma de alocação.

OBJETIVOS

- Estudar os principais aspectos envolvidos na utilização de ponteiros.
- Entender como é feita a alocação e desalocação de memória em C++.
- Compreender os diversos tipos de operadores envolvidos com ponteiros.

4.1 Definição de ponteiro

Os endereços de memória podem ser armazenados e manipulados em linguagem C++ através de ponteiros. Quando declaramos uma variável e o programa é executado, o espaço necessário para o tipo da variável é alocado (reservado) na memória pelo sistema operacional como também recebe um nome, que é o nome da variável. Podemos guardar este endereço de memória em um ponteiro para que possa ser acessado diretamente.

Varias são as razões para o uso de ponteiros. Mizrahi (2006) cita as seguintes:

- Permitir de forma real que as funções alterem os parâmetros que recebe.
- Permitir uma forma mais conveniente de passar matrizes e string para funções.
- Facilitar a utilização de matrizes utilizando a movimentação de ponteiros para acesso de seus elementos.
- Permitir a criação e manipulação de estruturas de dados complexas que necessitam conter referências umas para outras, tais como, listas encadeadas, árvores binárias , etc.

Segundo Mizrahi (2006), os ponteiros podem ser visto como um modo simbólico de representar um endereço de memória. Ou seja, o objetivo de um ponteiro é guarda um endereço de memória.

4.2 Operador de endereço, operador de indireção e operador seta

4.2.1 Operador de endereço

No desenvolvimento de programas, constantemente há necessidade de armazenar dados utilizando as variáveis. As variáveis são armazenadas na memória do computador. Podemos imaginar o computador como se fosse um imenso armário com milhares de gavetas. Cada gaveta seria uma posição de memória onde podemos guardar alguma informação. Estas gavetas têm uma identificação única, para que possa ser localizada. Esta identificação única é o endereço de memória.

O programa abaixo possui duas seções. Na primeira, as variáveis são declaradas e na segunda recebem valores.

```
#include <iostream>
using namespace std;

void main(){

    // SEÇÃO 1: Declaração de variáveis
    char Tipo;     //M: Masculino   F: Feminino
    int Idade;
    float Preco;

    // SEÇÃO 2: Atribuição de valores
    Tipo = 'M';
    Idade = 30;
    Preco = 145.50;

    system("pause > null");
}
```

A figura 4.1 ilustra a primeira seção. A variável, por exemplo, Tipo foi criada e colocada na posição de memória de endereço 12. Idade foi colocada no endereço 16 e Preço no 20 (figura 4.1).

Valor						
Nome	Tipo	Idade	Preço			
Endereço	1004	1008	1012	1016	1020	1024

Figura 4.1 – Declaração de variáveis.

Em seguida, as variáveis receberam valores (figura 4.2).

Valor	M	30	145.50			
Nome	Tipo	Idade	Preço			
Endereço	1004	1008	1012	1016	1020	1024

Figura 4.2 – Variáveis com valores atribuídos.

Em linguagem C++, podemos obter o endereço de memória de uma variável utilizando o operador de endereço &.

```
cout << "Valor de Idade: " << Idade << endl;
cout << "Endereço de Idade: " << &Idade << endl;

cout << "Valor de Preço: " << Preco << endl;
cout << "Endereço de Preço: " << &Preco << endl;
```

Os endereços abaixo é o resultado impresso na tela. Estes endereços podem variar pelo fato do sistema operacional alocar a memória disponível levando em conta vários fatores, tais como, os programas em execução no momento, o espaço disponível, tamanho do tipo da variável, etc.

```
Valor de Idade: 30
Endereço de Idade: 001CF748

Valor de Preco: 145.5
Endereço de Preco: 001CF73C
```

4.2.2 Operador de indireção

A declaração de ponteiro pode ser feita da mesma forma que a de uma variável: definimos o tipo e o nome da variável precedida do caractere *. O nome do ponteiro segue as mesmas regras utilizadas para nomes de variáveis.

```
int *ptr;
```

A declaração permite que o ponteiro `ptr` armazene qualquer endereço de memória referente a um valor inteiro. Para utilizar o ponteiro utilizamos dois operadores unários: & e *.

O operador de endereço &, indica o endereço de, permite atribuir ao ponteiro o endereço de uma variável. Já o operador de indireção *, indica conteúdo de, permite acessar o conteúdo de um endereço armazenado em um ponteiro. Para exemplificar, efetuaremos um simples programa.

```
#include <iostream>
using namespace std;

void main(){
    int Idade; // Variável inteiro
    int *ptr;  // Variável ponteiro para um inteiro

    Idade = 25;
    ptr = &Idade; // Recebe o endereço de Idade

    *ptr = 30;

    cout << "Idade: " << Idade << endl << endl;
    cout << "Endereço de ptr: " << ptr << endl;
    cout << "Valor de ptr: " << *ptr << endl;

    system("pause > null");
}
```

A saída do programa será:

```
Idade: 30

Endereço de ptr: 0038FABC
Valor de ptr: 30
```

A seguir, será analisada cada parte do programa. Primeiramente, declaram-se as variáveis. Isto faz com que o espaço na memória seja alocado para comportar os tipos inteiros da variável e do ponteiro.

```
int Idade; // Variável inteiro
   int *ptr;  // Variável ponteiro para um inteiro
```

Valor						
Nome		Idade	ptr			
Endereço	1004	1008	1012	1016	1020	1024

Figura 4.3 – Declaração das variáveis.

Em seguida, a variável Idade recebe o valor 5 e o ponteiro recebe o endereço de memória da variável Idade. Neste ponto, é importante notar que, foi colocado o operador & na variável Idade para que o seu endereço possa ser atribuído ao ponteiro. Este tipo de atribuição pode ser dita como, `ptr` aponta para Idade, por isso tem-se o nome de ponteiro.

```
Idade = 25;
ptr = &Idade; // Recebe o endereço de Idade
```

Valor		25	1008			
Nome		Idade	ptr			
Endereço	1004	1008	1012	1016	1020	1024

Figura 4.4 – Atribuição de valores.

Como o ponteiro `ptr` aponta para a variável Idade, e seu valor for alterado, a variável Idade receberá indiretamente esta alteração. O caractere *, antes do ponteiro, indica que o ponteiro ira receber um valor e não um endereço. O valor recebido, 30, será colocado no endereço apontado pelo ponteiro, neste caso é o endereço 1008, que é o mesmo da variável Idade. Concluindo, uma variável acessa diretamente um valor enquanto o ponteiro acessa indiretamente um valor. A referência de um valor, através de um ponteiro é denominado indireção.

```
*ptr = 30;
```

Valor		30	1008			
Nome		Idade	ptr			
Endereço	1004	1008	1012	1016	1020	1024

Figura 4.5 – Alterando valor de um ponteiro.

Da mesma forma que uma variável pode atribuir seu valor para outra variável, um ponteiro pode atribuir seu endereço para outro ponteiro.

```
#include <iostream>
using namespace std;

void main(){
    system("chcp 1252 > nul");

    int Idade; // Variável inteiro
    int *ptrA, *ptrB;  // Variável ponteiro para um inteiro

    Idade = 25;
    ptrA = &Idade; // Aponta ptrA para Idade

    ptrB = ptrA;   // ptrB aponta para o mesmo endereço que ptrA

    cout << "Idade: " << Idade << endl << endl;

    cout << "Endereço de ptrA: " << ptrA << endl;
    cout << "Valor de ptrA: " << *ptrA << endl << endl;

    cout << "Endereço de ptrB: " << ptrB << endl;
    cout << "Valor de ptrB: " << *ptrB << endl;

    system("pause > null");
}
```

A saída do programa será:

```
Idade: 25

Endereço de ptrA: 004CF86C
Valor de ptrA: 25

Endereço de ptrB: 004CF86C
Valor de ptrB: 25
```

Primeiramente o ponteiro ptrA recebe o endereço de Idade. Em seguida, atribui seu endereço a ptrB. Isto faz com que tanto ptrA, como ptrB apontem para o mesmo endereço de memória, neste caso, o endereço da variável Idade. Isto indica que a variável Idade pode ser alterada indiretamente, tanto por ptrA, como ptrB.

```
ptrA = &Idade; // ptrA aponta para Idade
ptrB = ptrA;   // ptrB aponta para o mesmo endereço que ptrA
```

Valor		30	1008	1008		
Nome		Idade	ptrA	ptrB		
Endereço	1004	1008	1012	1016	1020	1024

Figura 4.6 – Atribuição de endereços entre ponteiros.

O valor de um ponteiro pode ser atribuída a uma variável. No exemplo abaixo, ao ponteiro tem acesso a altura do aluno A. Esta altura é atribuído ao aluno B utilizando o operador de indireção *.

```
#include <iostream>
using namespace std;

void main(){
    system("chcp 1252 > nul");

    float AlturaAlunoA, AlturaAlunoB;
    float *ptrAltura;

    AlturaAlunoA = 1.70;
    ptrAltura = &AlturaAlunoA; // Recebe o endereço de AlturaAlunoA

    AlturaAlunoB =  *ptrAltura;

    cout << "Altura aluno A: " << AlturaAlunoA << endl;
    cout << "Altura aluno B: " << AlturaAlunoB << endl;
```

```
    system("pause > null");
}
```

A saída do programa será:

```
Altura aluno A: 1.7
Altura aluno B: 1.7
```

Um ponto importante, que se deve ter atenção, com relação a ponteiros é a questão da sua inicialização. Um ponteiro deve ser inicializado antes de ser utilizado. Isto quer dizer que o ponteiro deve estar apontando para alguma posição válida antes de receber algum valor. No exemplo a seguir, o ponteiro é inicializado quando recebe o endereço de Idade.

```
#include <iostream>
using namespace std;
void main(){
    int Idade; // Variável inteiro
    int *ptr;  // Variável ponteiro para um inteiro
    Idade = 25;
    ptr = &Idade; // INICIALIZAÇÃO DO PONTEIRO
    *ptr = 30;
    system("pause > null");
}
```

CONEXÃO

Aprofunde seus conhecimentos sobre ponteiros em: http://www.tiexpert.net/programacao/c/ponteiros.php.

Um ponteiro que não é inicializado pode causar sérios problemas no programa. O ponteiro nesta situação pode estar lendo ou escrevendo em uma posição inválida de memória. No caso da leitura, pode estar lendo uma informação inválida. O pior pode acontecer na escrita. O código abaixo ilustra um valor, 30, sendo atribuído a um ponteiro, `ptr`, que não foi inicializado.

```
#include <iostream>
   using namespace std;

   void main(){
        int Idade; // Variável inteiro
        int *ptr;  // Variável ponteiro para um inteiro

        Idade = 25;

        // Atribuição de valor a um ponteiro NÃO INICIALIZADO
        *ptr = 30;

        system("pause > null");
   }
```

O ponteiro `ptr` pode estar escrevendo, o valor recebido, sobre trechos de dados, ou sobre o próprio código, o que pode acarretar um fechamento inesperado do programa. Este problema da escrita, em determinadas situações, pode ocorrer de forma intermitente, dificultando a sua localização. Dessa forma, é importante ficar atento a inicialização do ponteiro antes de utilizá-lo.

4.2.3 Aritmética de ponteiros

As operações aritméticas permitidas para ponteiros são duas: adição (incremento) e subtração (decremento). IMPORTANTE: não é permitido multiplicar ou dividir ponteiros.

```
ptrNota++; // Incremento do endereço do ponteiro
ptrNota--; // Decremento do endereço do  ponteiro
```

A operação de incremento faz com que o ponteiro acesse o próximo endereço de memória do tipo que ele foi criado. Assim, um ponteiro do tipo `int`, quando incrementado, apontará para o próximo endereço `int` da memória.

```
#include <iostream>
using namespace std;
```

```
void main(){
    int Nota;
    int *ptrNota;

    ptrNota = &Nota; // Inicializa o ponteiro

    cout << "Endereço 1: " << ptrNota << endl;

    ptrNota++;
    cout << "Endereço 2: " << ptrNota << endl;

    ptrNota++;
    cout << "Endereço 3: " << ptrNota << endl;

    system("pause > null");
}
```

A saída do programa será:

```
Endereço 1: 0039FD60
Endereço 2: 0039FD64
Endereço 3: 0039FD68
```

Como o ponteiro foi criado do tipo `int`, e o tipo `int` ocupa 4 bytes da memória, a cada incremento, o ponteiro avançou 4 bytes. O incremento é semelhante a soma de uma posição ao endereço de memória. As duas operações abaixo fazem a mesma coisa.

```
ptrNota++;
ptrNota = ptrNota + 1;
```

Dessa forma, é possível incrementar ou decrementar um ponteiro em mais de uma posição por vez. No exemplo a seguir, o ponteiro irá avançar 5 posições de memória.

```
#include <iostream>
   using namespace std;
   void main(){
        int Nota;
        int *ptrNota;

        ptrNota = &Nota; // Inicializa o ponteiro

        cout << "Endereço 1: " << ptrNota << endl;

        ptrNota = ptrNota + 5; // incrementa 5 posições do tipo int

        cout << "Endereço 2: " << ptrNota << endl;
        system("pause > null");
   }
```

A saída do programa será:

```
Endereço 1: 0018FAC4
Endereço 2: 0018FAD8
```

Ponteiros, vetores e matrizes

Matrizes e vetores são variáveis que ocupam endereço sequencial na memória. A quantidade de memória alocada depende do tipo da matriz. Por exemplo, o tipo `int`, ocupa 4 *bytes* de memória. Dessa forma, o compilador ao criar uma matriz `int NotasAlunos[20][3]`, alocará espaço sequencial de 240 *bytes* de memória.

```
20 x 3 x 4bytes = 240 bytes.
```

Matrizes e vetores permitem armazenar uma coleção de variáveis do mesmo tipo. Enquanto a matriz possui várias dimensões, o vetor possui apenas uma dimensão.

A linguagem C++ permite que qualquer operação que possa ser feita com índices de uma matriz ou vetor, possa também ser feita da mesma forma com vetores. Inicialmente, faremos um exemplo criando um vetor para receber as notas bimestrais de um aluno. Em seguida, utilizando o operador de endereço &, apontamos o ponteiro para o endereço da nota do terceiro bimestre do aluno.

```
#include <iostream>
using namespace std;

void main(){

    float Nota[3];
    float *ptrPrimBimestre;

    Nota[0] = 7.8;
    Nota[1] = 9.0;
    Nota[2] = 7.0;
    Nota[3] = 9.5;

    ptrPrimBimestre = &Nota[2]; // Inicializa o vetor

    cout << "Nota 3° bimestre: " << *ptrPrimBimestre << endl;

    system("pause > null");
}
```

Para que seja possível o ponteiro acessar todos os elementos do vetor, o programa precisa ser alterado da seguinte forma:

```
#include <iostream>
using namespace std;

void main(){
    float Nota[3];
    float *ptrNota;
```

```
    Nota[0] = 7.8;
    Nota[1] = 9.0;
    Nota[2] = 7.0;
    Nota[3] = 9.5;

     // O ponteiro aponta para o primeiro elemento de Nota
    ptrNota = Nota;

    cout << "Nota 1° bimestre: " << ptrNota[0] << endl;
    cout << "Nota 2° bimestre: " << ptrNota[1] << endl;
    cout << "Nota 3° bimestre: " << ptrNota[2] << endl;
    cout << "Nota 4° bimestre: " << ptrNota[3] << endl;

    system("pause > null");
}
```

A impressão de cada nota foi feita com o ponteiro utilizando a notação de vetor. Notação de vetor é quando se utiliza os colchetes []. Outro ponto a ser observado é com relação ao trecho:

```
// O ponteiro aponta para o primeiro elemento de Nota
ptrNota = Nota;
```

O ponteiro esta recebendo o endereço do primeiro elemento do vetor. É importante notar que neste caso não foi utilizado o operador de endereço &. Em C++, vetores e matrizes são tratadas internamente como se fossem ponteiros. O nome de um vetor aponta permanentemente para o seu primeiro elemento. Uma alteração possível, para a atribuição do endereço do primeiro elemento do vetor para o ponteiro, é expresso a seguir. Os dois códigos resultam na mesma coisa,

```
ptrNota = Nota;
ptrNota = &Nota[0];
```

Apesar de vetores e matrizes poderem ser consideradas como ponteiro, não é possível alterar o seu endereço. O endereço de uma matriz é permanente.

Dessa forma a operação, NotaB = NotaA, do código abaixo, não é válida.

```
#include <iostream>
using namespace std;

void main(){
    float NotaA[2], NotaB[2];

    NotaA[0] = 7.8;
    NotaA[1] = 9.0;

    NotaB = NotaA; // OPERAÇÃO INVÁLIDA

    system("pause > null");
}
```

Apesar de ser possível utilizar a notação de vetores com ponteiros, o ponteiro tem sua própria notação, denominada notação de ponteiro. O próximo exemplo utilizará o ponteiro, tanto com a notação de vetor como a notação de ponteiro. Dessa forma podemos efetuar a comparação entre as duas notações.

```
#include <iostream>
using namespace std;

void main(){
    system("chcp 1252 > nul");

    float Nota[3];
    float *ptrNota;

    Nota[0] = 7.8;
    Nota[1] = 9.0;
    Nota[2] = 7.0;
    Nota[3] = 9.5;

    ptrNota = &Nota[0]; // Inicializa o vetor apontando para Nota
```

```
    cout << "Utilizando notação de vetor " << endl;
    cout << "Nota 1° bimestre: " << ptrNota[0] << endl;
    cout << "Nota 2° bimestre: " << ptrNota[1] << endl;
    cout << "Nota 3° bimestre: " << ptrNota[2] << endl;
    cout << "Nota 4° bimestre: " << ptrNota[3] << endl << endl;

    cout << "Utilizando notação de ponteiro " << endl;
    cout << "Nota 1° bimestre: " << *(ptrNota) << endl;
    cout << "Nota 2° bimestre: " << *(ptrNota + 1) << endl;
    cout << "Nota 3° bimestre: " << *(ptrNota + 2) << endl;
    cout << "Nota 4° bimestre: " << *(ptrNota + 3) << endl;

    system("pause > null");
}
```

A saída do programa será:

```
Utilizando notação de vetor
Nota 1° bimestre: 7.8
Nota 2° bimestre: 9
Nota 3° bimestre: 7
Nota 4° bimestre: 9.5

Utilizando notação de ponteiro
Nota 1° bimestre: 7.8
Nota 2° bimestre: 9
Nota 3° bimestre: 7
Nota 4° bimestre: 9.5
```

Quando o ponteiro recebe o endereço da variável, ptrNota = &Nota[0]; está apontando para o primeiro endereço do, que é onde está o seu primeiro elemento. Assim para imprimir o valor do primeiro elemento, basta usar a notação de ponteiro *(ptrNota). Para acessar o segundo elemento, é necessário ir para o segundo endereço. Isto é feito somando 1 ao endereço do ponteiro, *(ptrNota + 1). Independente da notação utilizada, o resultado final foi o mesmo. As duas expressões abaixo têm o mesmo valor.

```
ptrNota[3]
   *(ptrNota + 3)
```

Como o ponteiro ptrNota aponta para o endereço do primeiro elemento do vetor, para acessar o valor de outro elemento, deve-se incrementar o endereço do ponteiro.

4.2.4 Ponteiros e funções

Uma função pode ter ponteiros em seus parâmetros. Para passar uma variável para um ponteiro, que é parâmetro de uma função, esta variável deve ser precedida pelo operador de endereço &. O exemplo a seguir, reajusta o valor de um produto de acordo com a taxa especificada. O cálculo é feito passando o endereço da variável PrecoProduto para o ponteiro *Preco. Consequentemente, qualquer alteração feita em *Preco refletirá no valor de PrecoProduto.

```
#include <iostream>
using namespace std;

void AjustarPreco(float *Preco, float Taxa);

void main(){
    float PrecoProduto;

    PrecoProduto = 500.00;

    AjustarPreco(&PrecoProduto, 20);

    cout << "Preço Reajustado: " << PrecoProduto << endl;

    system("pause > null");
}
```

```
void AjustarPreco(float *Preco, float Taxa){
       float ValorReajuste;

       ValorReajuste = (*Preco * Taxa) / 100; // Calcula o valor do
reajuste
       *Preco = *Preco + ValorReajuste; // Reajusta o valor
   }
```

A saída do programa será:

```
Preço Reajustado: 600
```

No exemplo a seguir, será passando um vetor para um parâmetro ponteiro de uma função. O programa apenas atribui um mesmo valor a cada elemento do atributo. Note que o vetor PagtoMes está sendo passado para a função sem o operador de endereço &. Lembrando que o nome do vetor, internamente contém o endereço do seu primeiro elemento.

```
   #include <iostream>
   using namespace std;

   void GerarValores(float *PagtoM, float VlrPagto);

   void main(){
      int Ind;

       // Armazena o código do aluno e sua nota bimestral
       float PagtoMes[12];

       GerarValores(PagtoMes, 150.80);

       for(Ind = 0; Ind < 12; Ind++){
              cout << "Pagamento Mês " << Ind << ": " <<PagtoMes[Ind]
<< endl;
      }
              system("pause > null");

   }
```

```
void GerarValores(float *PagtoM, float VlrPagto){
    int Ind;

    for(Ind = 0; Ind < 12; Ind++){
           *(PagtoM + Ind) = VlrPagto;
    }
}
```

Note o trecho abaixo, retirado do programa. Está sendo utilizada aritmética de ponteiros para acessar os elementos do ponteiro.

```
*(PagtoM + Ind) = VlrPagto;
```

Para passar matrizes para uma função, alguns detalhes devem ser observados. O programa a seguir tem como objetivo guardar informações sobre cursos. Para isso foi criado uma matriz com os dados do número da turma, quantidade de alunos e quantidade de disciplina. A matriz é passada para o ponteiro *TurmaAD, parâmetro da função Imprimir(...).

```
#include <iostream>
using namespace std;

void Imprimir(int *TurmaAD);

void main(){
    // Armazena: Turma, Qtde Aluno, Qtde Disciplinas
    int TurmaALunoDisc[3][3];

    TurmaALunoDisc[0][0] = 100;  // Nro da turma
    TurmaALunoDisc[0][1] = 20;   // Qtde de alunos
    TurmaALunoDisc[0][2] = 6;    // Qtde Discliplinas

    TurmaALunoDisc[1][0] = 200;  // Nro da turma
    TurmaALunoDisc[1][1] = 18;   // Qtde de alunos
    TurmaALunoDisc[1][2] = 5;    // Qtde Discliplinas

    TurmaALunoDisc[2][0] = 300;  // Nro da turma
    TurmaALunoDisc[2][1] = 15;   // Qtde de alunos
    TurmaALunoDisc[2][2] = 4;    // Qtde Discliplinas
```

```
        Imprimir(&TurmaALunoDisc[0][0]);

        system("pause > null");
    }

    void Imprimir(int *TurmaAD){
      int Lin;

      for(Lin = 0; Lin < 3; Lin++) {
        cout << "Nro da turma: " << *(TurmaAD + (Lin * 3)) << endl;
        cout << "Qtde de alunos: " << *(TurmaAD + (Lin * 3) + 1) << endl;
        cout << "Qtde Discliplinas: " << *(TurmaAD + (Lin * 3) + 2) <<
endl cout << endl << endl;
      }
    }
```

A saída do programa será:

```
Nro da turma: 100
Qtde de alunos: 20
Qtde Discliplinas: 6

Nro da turma: 200
Qtde de alunos: 18
Qtde Discliplinas: 5

Nro da turma: 300
Qtde de alunos: 15
Qtde Discliplinas: 4
```

O detalhe deste programa está na passagem da matriz para a função. Como a matriz possui duas dimensões, na linguagem C++ é necessário colocar o operador de endereço & e também os colchetes indicando os primeiros elementos de cada dimensão.

```
Imprimir(&TurmaALunoDisc[0][0]);
```

4.2.5 Operador seta

O operar seta, "->", é utilizado quando o ponteiro está acessando uma estrutura. Esta notação é diferente do acesso de uma estrutura por uma variável, que neste caso utiliza o operador ponto. O operar seta, "->", permite que o ponteiro acesse os elementos de uma estrutura.

```
#include <iostream>
using namespace std;

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};

void main(){
    DADOS_ALUNO AlunoA ;
    DADOS_ALUNO *ptrAluno;

    cout << "Digite o código do aluno: ";
    cin >> AlunoA.CodAluno;
    cout << "Digite o nome do aluno: ";
    cin >> AlunoA.Nome;
    cout << "Digite a turma: ";
    cin >> AlunoA.Turma;

    ptrAluno = &AlunoA;

    cout << "Código do Aluno: " << ptrAluno->CodAluno <<endl;
    cout << "Nome: " << ptrAluno->Nome <<endl;
    cout << "Turma: " <<  ptrAluno->Turma <<endl;


    system("pause > null");
}
```

O operar seta, “->”, permite acessar um membro da estrutura para obter ou atribuir um valor. A atribuição de um valor ao membro da estrutura é feita como no exemplo a seguir.

```
ptrAluno->CodAluno = 150;
```

A passagem de uma estrutura para um função através de um parâmetro ponteiro é exemplificado a seguir.

```
#include <iostream>
using namespace std;

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};

void Imprimir(DADOS_ALUNO *ptrAluno);

void main(){
    DADOS_ALUNO AlunoA ;

    cout << "Digite o código do aluno: ";
    cin >> AlunoA.CodAluno;
    cout << "Digite o nome do aluno: ";
    cin >> AlunoA.Nome;
    cout << "Digite a turma: ";
    cin >> AlunoA.Turma;

    Imprimir(&AlunoA);

    system("pause > null");
}
```

```
void Imprimir(DADOS_ALUNO *ptrAluno){
    cout << "Código do Aluno: " << ptrAluno->CodAluno <<endl;
    cout << "Nome: " << ptrAluno->Nome <<endl;
    cout << "Turma: " <<  ptrAluno->Turma <<endl;
}
```

4.3 Alocação e desalocação de memória

Ao desenvolver um programa, principalmente utilizando estrutura de dados, é muito difícil prever com antecedência quantos elementos terá esta estrutura. Por exemplo, ao criar uma pilha utilizando vetores, qual seria a dimensão de elementos desta pilha? Se a pilha for dimensionada para receber 200 elementos e durante a sua utilização, não for necessário mais que 50 elementos, estaremos desperdiçando memória. Por outro lado, poderia ser necessário mais que os 200 especificados inicialmente.

CONEXÃO

Entenda um pouco mais sobre alocação e desalocação de memória em: http://www.pontov.com.br/site/cpp/46-conceitos-basicos/57-matrizes-dinamicas.

Este tipo de problema pode ser solucionado alocando memória conforme for necessário durante a execução do programa. Em C++ a alocação de memória, em tempo de execução, é feita utilizando o operador `new`. O operador `new` aloca memória do sistema operacional e retorna um ponteiro para o primeiro *byte* do bloco de memória alocado. Para liberar a memória alocada é utilizado o operador delete.

```
#include <iostream>
using namespace std;

void main(){
    // Aloca memória
```

```
    int *ptrNroAlunos = new int;

    *ptrNroAlunos = 150;

    cout << "Quantidade de alunos: " << *ptrNroAlunos ;

    //Desaloca memória
    delete ptrNroAlunos;

    system("pause > null");
}
```

O exemplo a seguir aloca memória para armazenar dados de três alunos.

```
#include <iostream>
using namespace std;

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
};

void main(){
    int Ind;

    // Aloca memória
    DADOS_ALUNO *prtAluno = new DADOS_ALUNO[3];

    prtAluno[0].CodAluno = 10;
    strcpy(prtAluno[0].Nome, "Maria");
    prtAluno[0].Turma = 320;

    prtAluno[1].CodAluno = 20;
    strcpy(prtAluno[1].Nome, "José");
    prtAluno[1].Turma = 320;
```

```
    prtAluno[2].CodAluno = 30;
    strcpy(prtAluno[2].Nome, "João");
    prtAluno[2].Turma = 320;

    for(Ind = 0; Ind < 3; Ind++){
            cout << "Código do aluno: " << prtAluno[Ind].CodAluno << endl;
            cout << "Nome do aluno: " << prtAluno[Ind].Nome << endl;
    cout << "Turma: " << prtAluno[Ind].Turma << endl;
    cout << endl << endl;

    }

    //Desaloca memória
    delete []prtAluno;

    system("pause > null");
}
```

A alocação de espaço para o vetor é feita com base na estrutura criada. O trecho de código abaixo solicitou ao programa que alocasse espaço para conter três estruturas do tipo DADOS_ALUNO.

```
DADOS_ALUNO *prtAluno = new DADOS_ALUNO[3];
```

Uma vez o espaço alocado, o ponteiro utiliza a notação vetor para acessar os elementos da matriz.

```
prtAluno[0].CodAluno = 10;
strcpy(prtAluno[0].Nome, "Maria");
prtAluno[0].Turma = 320;
```

ATIVIDADES

01. Analise o código abaixo e indique qual o valor final da variável x?

```
int main() {
int x, y, *z;
x=10;
y = 20;
z = &x;
x++;
*z = x + y;
(*z)++;
cout <<  x << endl;
}
```

a) 30
b) 32
c) 10
d) 11
e) 31

02. Ao enviar o endereço de uma variável como parâmetro de entrada de uma função, este endereço deve ser recebido por meio de:

a) passagem de valor.
b) passagem de inferência.
c) inferência.
d) ponteiro.
e) registro.

03. De acordo com o material, quais são os comandos para alocar e desalocar dinamicamente memória na linguagem C++?

a) aloc e maloc.
b) insert e delete.
c) insert e remove.
d) new e delete.
e) new e remove.

REFLEXÃO

Neste capítulo estudamos o que são e para que serve os ponteiros. Para melhor entender os ponteiros iniciamos vendo os seus operadores de endereço, de indireção e operador seta. Em seguida, abordamos a aritmética de ponteiros e criamos programas para demostrar seu funcionamento. Aprendemos como utilizar ponteiros com vetores e matrizes. Por fim, vimos os mecanismos presentes na linguagem C++ para alocação e desalocação de memória.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de sistemas operacionais e demais assuntos deste capítulo, consulte as sugestões de links abaixo:

Capítulo 11 do livro: MIZRAHI, V. V. Treinamento em Linguagem C++: Módulo 2. 2ª ed. São Paulo, Editora Prentice-Hall, 2006.

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M., & DEITEL, P. J. **C++ Como Programar.** 3ª ed. Porto Alegre: Bookman, 2001.

MIZRAHI, V. V. **Treinamento em Linguagem C++: Módulos 1 e 2.** 2ª ed. São Paulo: Prentice Hall, 2006.

SCHILDT, H. **C Completo e Total.** 3ª ed. São Paulo: Makron Books Editora Ltda, 1996.

TANENBAUM, A. M., LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de Dados Usando C.** São Paulo: Makron Books, 1995.

5

Listas Lineares Encadeadas

Estudamos no capítulo três a implementação de estruturas utilizando vetores e matrizes. Esta abordagem traz algumas limitações, a principal é conhecer com antecedência a quantidade de elementos da estrutura. Outro fator, é que vetores são recomendados para estruturas com poucos elementos e, a cada dia que passa, os programas têm como requisitos, manusear uma quantidade maior de dados.

A solução destes problemas está na implementação de estruturas de dados utilizando ponteiros e alocação dinâmica. Assim é possível utilizar melhor e de forma mais eficiente a memória do computador para comportar a necessidade crescente de armazenamento dos dados.

OBJETIVOS

- Implementar as estruturas de dados utilizando ponteiros e alocação de memória;
- Entender as principais formas de acesso às estruturas de dados;
- Compreender os diversos modos de implementação de estruturas dinâmicas.

5.1 Listas lineares Simplesmente Encadeadas

Estudamos no capítulo 1, as particularidades de cada uma das estruturas de dados: lista, pilha e fila. Avançamos nossos estudos implementando estas estruturas utilizando vetores. Uma limitação nas abordagens utilizando vetores é a necessidade de definir, antecipadamente, a quantidade de elementos das estruturas. Consequentemente, pode haver um subdimensionamento do vetor e assim, não atender a demanda esperada ou um sobredimensionamento, causando desperdício de memória.

A solução está em utilizar ponteiros na implementação das estruturas. Utilizando ponteiros é possível as estruturas crescerem, tendo como única limitação a disponibilidade de memória para a aplicação.

Os ponteiros permitem que os nós ocupem espaços aleatórios na memória, diferentemente dos vetores onde os nós ocupam espaços contíguos. Dessa forma, os nós necessitam conhecer a localização do próximo nó (endereço) da sequência. Isto é feito, adicionando um novo campo ao nó, que armazenará este endereço. Segundo Villas et al. (1993) a está novo campo é dado o nome de *link.*

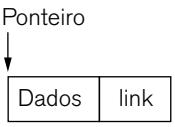


Figura 5.1 – Diagrama do nó. Fonte: Villas et al. (1993).

A partir desta formatação do nó é possível criar uma lista simplesmente encadeada conforme a ilustração a seguir. A lista pode ou não conter um nó especial, para marcador do início da lista, denominado Nó Cabeça.

> O nó cabeça também é conhecido como nó de cabeçalho, ***header*** ou sentinela. O nó cabeça pode conter informações gerais a respeito da estrutura de dados.

Apesar de utilizar a mesma estrutura dos demais nós, o nó cabeça não tem os dados das variáveis membros preenchidas.

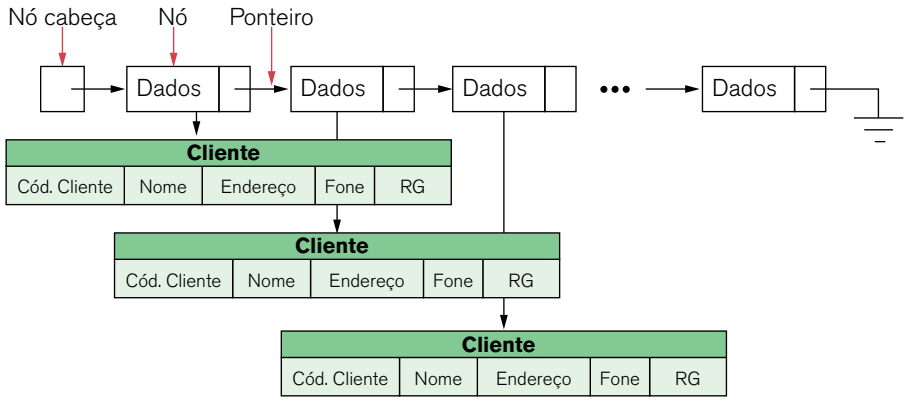


Figura 5.2 – Lista simplesmente encadeada.

O fato de cada nó utilizar uma informação adicional, o *link*, utiliza mais espaço de memória em comparação aos vetores. Por outro lado, além de não precisar dimensiornar a lista antecipadamente, não é necessário movimentar os elementos dos vetores ao inserir ou excluir um elemento. As operações de inserção e remoção de um nó são ilustradas a seguir. A inserção de um novo nó, basta atualizar os valores dos *links*.

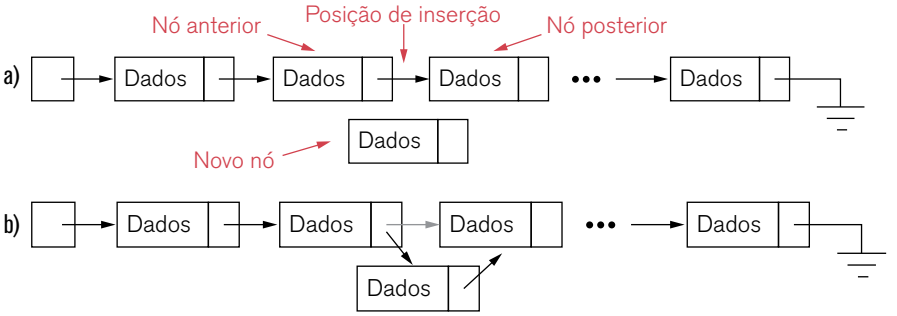


Figura 5.3 - Inserção de um nó.

Para efetuar a remoção de um nó, também, basicamente atualizamos os endereços armazenados no *link*.

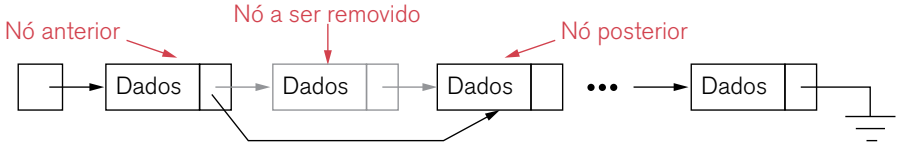


Figura 5.4 - Remoção de um nó.

5.2 Operações com listas lineares simplesmente encadeadas

Incialmente, iremos desenvolver um código que cria uma lista simplesmente encadeada com três nós: nó cabeça, primeiro nó e segundo nó. O objetivo e deixar o mais claro possível a criação de uma lista encadeada. Em seguida será feita a análise de cada trecho do programa. Por fim, com base neste código, serão desenvolvidas funções com as operações básicas em lista encadeada utilizando ponteiros.

```
#include <iostream>
#include "conio.h"

using namespace std;

struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;

    struct DADOS_ALUNO * ptrLink;
};

void main(){
    int Ind;
    struct DADOS_ALUNO *ptrCabeca;
    struct DADOS_ALUNO *ptrPrimeiraNo, *ptrSegundoNo;
    struct DADOS_ALUNO *ptrAux;

    ptrCabeca = new DADOS_ALUNO;
    ptrCabeca->ptrLink = NULL;

    // PRIMEIRO NÓ: Cria um novo nó
    ptrPrimeiraNo = new DADOS_ALUNO;
    ptrPrimeiraNo->CodAluno = 10;
```

```
    strcpy(ptrPrimeiraNo->Nome, "José");
    ptrPrimeiraNo->Turma = 250;
    ptrPrimeiraNo->ptrLink = NULL;

    // Liga o primeiro nó ao nó cabeça
    ptrCabeca->ptrLink = ptrPrimeiraNo;

    // SEGUNDO NÓ: Cria um novo nó
    ptrSegundoNo = new DADOS_ALUNO;
    ptrSegundoNo->CodAluno = 20;
    strcpy(ptrSegundoNo->Nome, "Maria");
    ptrSegundoNo->Turma = 250;
    ptrSegundoNo->ptrLink = NULL;

    // Liga o segundo nó ao primeiro nó
    ptrPrimeiraNo->ptrLink = ptrSegundoNo;

    ptrAux = ptrCabeca->ptrLink;
    Ind = 1;

    while(ptrAux != NULL){
           cout << "Nó: " << Ind << endl;
           cout << "Código do Aluno: " << ptrAux->CodAluno << endl;
           cout << "Nome: " << ptrAux->Nome << endl;
           cout << "Turma: " << ptrAux->Turma << endl;
           cout << endl << endl;
           Ind++;
           ptrAux = ptrAux->ptrLink;
    }
    // Libera o espaço de memória
    delete ptrCabeca;
    delete ptrPrimeiraNo;
    delete ptrSegundoNo;

    system("pause>null");
}
```

A saída do programa será:

```
Nó: 1
Código do Aluno: 10
Nome: José
Turma: 250
Nó: 2
Código do Aluno: 20
Nome: Maria
Turma: 250
```

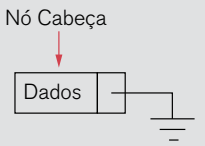
Efetuaremos a análise de cada trecho do código. Incialmente definimos a estrutura de dados. A estrutura é dividida logicamente em duas partes. A primeira contém os dados do nó: código do aluno, nome e turma. A segunda, contém o ponteiro, `* ptrLink,` que apontará para o próximo nó. Como o próximo nó da lista, será uma estrutura do tipo DADOS_ALUNO, o ponteiro foi declarado para permitir apontar também para uma estrutura do tipo DADOS_ALUNO.

```
struct DADOS_ALUNO{
     int CodAluno;
     char Nome[100];
     int Turma;

     struct DADOS_ALUNO * ptrLink;
};
```

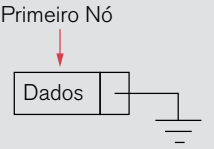
O ponto inicial da lista é o nó cabeça. Utilizando o operador `new` é alocado memória para conter a estrutura DADOS_ALUNO, necessária ao ponteiro *ptrCabeca. Apesar do nó ser criado como um tipo DADOS_ALUNO, os membros CodAluno, Nome e Turma não são utilizados. Será utilizado apenas o membro *ptrLink, que é o ponteiro que apontará para o próximo nó da lista. Como a lista ainda não possui outros nós, o ponteiro ptrCabeca->ptrLink aponta para NULL. NULL em linguagem C++ significa um endereço vazio. Isto é feito para que o ptrCabeca->ptrLink não aponte para um endereço inválido que possa corrompera o programa.

```
// Aloca memória para a estrutura
ptrCabeca = new DADOS_ALUNO;
// Aponta para um endereço vazio
ptrCabeca->ptrLink = NULL;
```



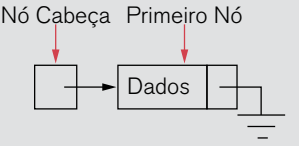
Nesta parte, é criado o primeiro nó. Utiliza-se o operador new para alocar o espaço necessário. Em seguida, atribuem se valores as variáveis membros da estrutura. Por fim, como o primeiro nó não tem um nó a sua frente, o ponteiro ptrPrimeiraNo->ptrLink aponta para um endereço vazio.

```
// Aloca memória para a estrutura
ptrPrimeiraNo = new DADOS_ALUNO;
// Atribui valores aos membros
ptrPrimeiraNo->CodAluno = 10;
strcpy(ptrPrimeiraNo->Nome, "José");
ptrPrimeiraNo->Turma = 250;
// Aponta para um endereço vazio
ptrPrimeiraNo->ptrLink = NULL;
```



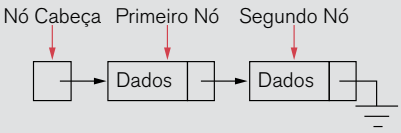
Até este ponto, foi feito apenas a criação e atribuição de valores ao nó ptrPrimeiraNo. O próximo passo é colocá-lo na lista. Nesta versão de lista, isto é feito, ligando-o ao último nó criado. O último nó criado, foi o nó cabeça. Assim é preciso ligar o nó cabeça ao primeiro nó. Para isso, alteramos o ponteiro ptrCabeca->ptrLink, que antes apontando para NULL, para apontar para o ponteiro ptrPrimeiraNo.

```
// Liga o primeiro nó ao nó cabeça
ptrCabeca->ptrLink = ptrPrimeiraNo;
```



O segundo nó, basicamente segue os mesmos passos descritos anteriormente para o primeiro nó. A diferença está que sua colocação na lista. Como vimos, o nó criado é ligado ao último nó da lista. Neste caso será o primeiro nó.

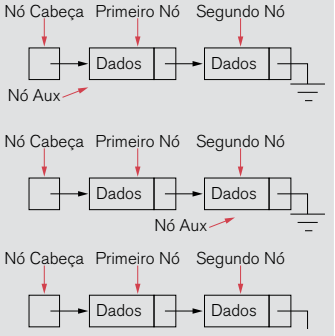
```
// Liga o segundo nó ao primeiro
nó
ptrPrimeiraNo->ptrLink =
ptrSegundoNo;
```



Uma vez a lista criada, imprimimos os seus elementos. Isto é feito utilizando um ponteiro auxiliar *ptrAux. O ponteiro *ptrAux aponta incialmente para o mesmo endereço do nó cabeça, ptrAux = ptrCabeca->ptrLink. Como o ponteiro ptrCabeca->ptrLink está apontando para o primeiro nó, também o ponteiro ptrAux apontará para o primeiro nó. O `loop while`, então exibe os dados do primeiro nó. Em seguida, dentro do `loop`, ptrAux recebe o endereço de onde o seu ponteiro ptrLink está apontando, ptrAux = ptrAux->ptrLink. Como ptrAux está apontando para o primeiro nó, ptrAux->ptrLink estará apontando para o segundo nó. Isto faz com que ptrAux passe a apontar para o segundo nó. Na terceira iteração, ptrAux estará apontando para NULL o que indica que chegou no fim da lista, parando então a impressão.

```
ptrAux = ptrCabeca->ptrLink;
Ind = 1;
while(ptrAux != NULL){
  cout <<"Nó: " << Ind << endl;
  cout <<"Código do Aluno: "<<
        ptrAux->CodAluno << endl;
  cout << "Nome: " << ptrAux->Nome << endl;
  cout << "Turma: " << ptrAux->Turma << endl;
  cout << endl << endl;

  Ind++;
  ptrAux = ptrAux->ptrLink;
}
```



Como nosso programa não efetuará mais nenhuma operação, os espaços de memória são desalocados através do comando delete.

```
// Libera o espaço de memória
delete ptrCabeca;
delete ptrPrimeiraNo;
delete ptrSegundoNo;
```

CONEXÃO

Leia mais sobre listas encadeadas em: http://www.ic.unicamp.br/~ra069320/PED/MC102/1s2008/Apostilas/Cap10.pdf.

A seguir serão apresentadas as operações básicas com lista simplesmente encadeada.

5.2.1 Criar lista

A função CriarLista(...) cria e retorna o primeiro nó. Este será o nó cabeça.

```
DADOS_ALUNO* CriarLista(){
    DADOS_ALUNO *ptrCab;

    ptrCab = new DADOS_ALUNO;
    ptrCab->ptrLink = NULL;

    return ptrCab;
}
```

A chamada da função fica da seguinte forma:

```
struct DADOS_ALUNO *ptrCabeca;

// Cria a lista
ptrCabeca = CriarLista();
```

5.2.2 Verificar lista vazia

A função a seguir recebe o nó cabeça e verifica se há algum nó na lista.

```
bool VerificarListaVazia(DADOS_ALUNO *ptrCab){
    if(ptrCab->ptrLink == NULL)
            return true;
    else
            return false;
}
```

5.2.3 Inserir um novo nó

A função Inserir(...) recebe como parâmetros: o nó cabeça, a posição onde o nó deve ser inserido e os dados referente ao aluno. Inicialmente é criado um nó e atribuído os dados às variáveis membros de sua estrutura. Em seguida, o nó é inserido em uma das seguintes posições: início da lista, em uma posição determinada ou no fim da lista. Se a lista está vazia, o parâmetro PosInserir é ignorado e o novo nó é inserido logo após o nó cabeça. Se é informado alguma valor em PosInserir, então o nó é será inserido nesta posição. Caso o valor de PosInserir seja zero, então o nó é inserido no fim da lista.

```
bool Inserir(DADOS_ALUNO *ptrCab, int PosInserir, int CodAluno,
            char Nome[], int Turma){

    DADOS_ALUNO *ptrNovo;
    DADOS_ALUNO *ptrAux;
    int Pos;

    //-----------------------------------------------------------
    // Cria o novo nó e atribui os dados às variáveis membros
    //-----------------------------------------------------------
    ptrNovo = new DADOS_ALUNO;
    if (ptrNovo == NULL){
            cout << "Memória insulficiente!";
            return false;
    }
```

```
ptrNovo->CodAluno = CodAluno;
strcpy(ptrNovo->Nome, Nome);
ptrNovo->Turma = Turma;
ptrNovo->ptrLink = NULL;


//-------------------------------------------------------------
// Se a lista estiver vazia, insere no início da lista
//-------------------------------------------------------------
if( VerificarListaVazia(ptrCab) ){

        // Liga o primeiro nó ao nó cabeça
        ptrCab->ptrLink = ptrNovo;
        return true;
}

//-------------------------------------------------------------
// Se não foi informada a posição, então insere no fim da lista
//-------------------------------------------------------------
if (PosInserir == 0)
{
        // Localiza o último nó
        ptrAux = ptrCab->ptrLink;
        while(ptrAux->ptrLink != NULL)
        ptrAux = ptrAux->ptrLink;

        ptrAux->ptrLink = ptrNovo;

        return true;
}

//-------------------------------------------------------------
// Insere na posição informada
//-------------------------------------------------------------
ptrAux = ptrCab;
Pos = 1;
```

```
        // Localiza a posição a ser inserida
        while((Pos < PosInserir) && (ptrAux != NULL)){
                ptrAux = ptrAux->ptrLink;
                Pos++;
        }

        if(ptrAux == NULL){
                cout << "Posição não encontrada!";
                return false;
        }

        ptrNovo->ptrLink = ptrAux->ptrLink;
        ptrAux->ptrLink = ptrNovo;
        return true;
}
```

5.2.4 Localizar um nó

A pesquisa na lista utiliza como parâmetro o código do aluno. O código é passado para a função PosicaoNo(...) que retorna a posição do nó, dentro da lista para o código informado.

```
int PosicaoNo(DADOS_ALUNO *ptrCab, int CodAluno){
        DADOS_ALUNO *ptrAux;
        int Posicao;

        //------------------------------------------------------------
        // Se a lista estiver vazia
        //------------------------------------------------------------
        if( VerificarListaVazia(ptrCab) ){
                cout << "Lista vazia!";
                return false;
        }
```

```
    ptrAux = ptrCab;
    Posicao = 0;

    // Localiza o nó
    while(ptrAux != NULL){
        if(ptrAux->CodAluno == CodAluno)
                break;

                Posicao++;
                ptrAux = ptrAux->ptrLink;
        }

        if(ptrAux == NULL){
                cout << "Código do aluno não encontrado!";
                return -1;
        }

        return Posicao;
    }
```

5.2.5 Obter o tamanho

A função ObterTamanho(...) recebe o nó cabeça e conta quantos nós existem na lista. O resultado é retornado pela função.

```
    int ObterTamanho(DADOS_ALUNO *ptrCab){
        DADOS_ALUNO *ptrAux;
        int Pos;

        //------------------------------------------------------------
        // Se a lista estiver vazia
        //------------------------------------------------------------
        if( VerificarListaVazia(ptrCab) ){
                cout << "Lista vazia!";
                return false;
        }
```

```
    Pos = 0;
    ptrAux = ptrCab->ptrLink;

    // Conta os nós
    while(ptrAux != NULL){
        Pos++;
        ptrAux = ptrAux->ptrLink;
}

    return Pos;
}
```

5.2.6 Exibir lista

A função Exibir(...) exibe todos os nós da lista.

```
void Exibir(DADOS_ALUNO *ptrCab){
    DADOS_ALUNO *ptrAux;
    int Ind;

    ptrAux = ptrCab->ptrLink;
    Ind = 1;

    while(ptrAux != NULL){
        cout << "Nó: " << Ind << endl;
        cout << "Código do Aluno: " << ptrAux->CodAluno << endl;
        cout << "Nome: " << ptrAux->Nome << endl;
        cout << "Turma: " << ptrAux->Turma << endl;
        cout << endl << endl;

        Ind++;
        ptrAux = ptrAux->ptrLink;
    }
}
```

5.2.7 Remover nó

A função Remover(...), remove o nó da lista em determinada posição. Uma vez o nó removido, seu espaço é liberado da memória através do comando delete.

```
bool Remover(DADOS_ALUNO *ptrCab, int PosRemover){
    DADOS_ALUNO *ptrAux;
    DADOS_ALUNO *ptrAnterior;
    int Pos;

    //-------------------------------------------------------------
    // Se a lista estiver vazia
    //-------------------------------------------------------------
    if( VerificarListaVazia(ptrCab) ){
           cout << "Lista vazia!";
           return false;
    }
    if(PosRemover < 1){
           cout << "Código do aluno não encontrado!";
           return false;
    }
    ptrAux = ptrCab;
    ptrAnterior = ptrCab;
    Pos = 0;

    // Localiza a posição a ser removida
    while((Pos < PosRemover) && (ptrAux != NULL)){
           ptrAnterior = ptrAux;
           ptrAux = ptrAux->ptrLink;
           Pos++;
    }
    if(ptrAux == NULL){
           cout << "Posição inválida!";
           return false;
    }
    ptrAnterior->ptrLink = ptrAux->ptrLink;
    delete ptrAux;
}
```

5.3 Operações com lista linear simplesmente encadeada, realizando aplicações

As listas simplesmente encadeadas podem ser utilizadas em diversos tipos de aplicações onde há necessidade de definir dinamicamente a quantidade de seus elementos. As aplicações são basicamente as definidas em listas sequenciais, a diferença aqui é que a lista é definida dinamicamente. Como exemplo de aplicações, citamos: cadastro de produtos, em um sistema de estoque, cadastro de veículos e clientes em um sistema de concessionária de veículos, sistemas bibliotecários, escolares, etc.

5.4 Pilha DINÂMICA

Neste tópico iremos estudar a pilha simplesmente encadeada. Sua principal característica está na forma como os nós são inseridos e removidos da pilha, que é conhecida como LIFO (*Last In, First Out* - Último a Entrar, Primeiro a Sair). Desta forma, o último nó que foi inserido na pilha será o primeiro a ser removido.

> A pilha simplesmente encadeada também é conhecida como pilha dinâmica. A pilha dinâmica é um tipo de lista simplesmente encadeada que tem como característica a inserção e remoção dos nós no topo da pilha.

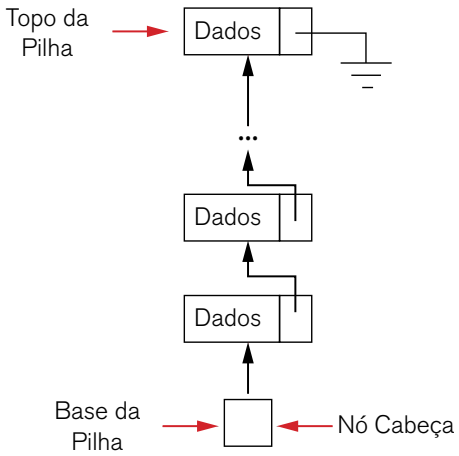


Figura 5.5 – Pilha simplesmente encadeada.

5.5 Operações com pilha dinâmica.

A seguir serão exibidas as operações básicas com pilha.

5.5.1 Criar uma pilha vazia

A função CriarPilha(...) cria e retorna um primeiro nó. Este primeiro nó, será o nó cabeça.

```
DADOS_ALUNO* CriarPilha(){
    DADOS_ALUNO *ptrCab;

    ptrCab = new DADOS_ALUNO;
    ptrCab->ptrLink = NULL;

    return ptrCab;
}
```

5.5.2 Verificar pilha vazia

A função a seguir recebe o nó cabeça e verifica se há algum nó na pilha.

```
bool VerificarPilhaVazia(DADOS_ALUNO *ptrCab){
    if(ptrCab->ptrLink == NULL)
            return true;
    else
            return false;
}
```

5.5.3 Empilhar (Push)

A função Empilhar(...) insere um novo nó no topo da pilha.

```
bool Empilhar(DADOS_ALUNO *ptrCab, int CodAluno, char Nome[], int
Turma){

    DADOS_ALUNO *ptrNovo;
    DADOS_ALUNO *ptrAux;
    int Pos;

    //-------------------------------------------------------
    // Cria o novo nó
    //-------------------------------------------------------
    ptrNovo = new DADOS_ALUNO;
    if (ptrNovo == NULL){
            cout << "Memória insulficiente!";
            return false;
    }

    ptrNovo->CodAluno = CodAluno;
    strcpy(ptrNovo->Nome, Nome);
    ptrNovo->Turma = Turma;
    ptrNovo->ptrLink = NULL;
```

```
    //-------------------------------------------------------
    // Insere no topo da pilha
    //-------------------------------------------------------
ptrAux = ptrCab;
while(ptrAux->ptrLink != NULL)
ptrAux = ptrAux->ptrLink;

ptrAux->ptrLink = ptrNovo;

return true;
}
```

5.5.4 Exibir o topo da pilha (Stacktop)

A função ExibirTopo(...) tem como objetivo exibir o nó do topo da pilha.

```
void ExibirTopo(DADOS_ALUNO *ptrCab){
    DADOS_ALUNO *ptrAux;

    ptrAux = ptrCab->ptrLink;

    // Localiza a posição do topo
    while(ptrAux->ptrLink != NULL){
    ptrAux = ptrAux->ptrLink;
    }

    cout << "Código do Aluno: " << ptrAux->CodAluno << endl;
    cout << "Nome: " << ptrAux->Nome << endl;
    cout << "Turma: " << ptrAux->Turma << endl;
    cout << endl << endl;
}
```

5.5.5 Exibir toda a pilha (Pop)

A função a seguir exibe todos os nós da pilha.

```
void Exibir(DADOS_ALUNO *ptrCab){
     DADOS_ALUNO *ptrAux;
     int Ind;

     ptrAux = ptrCab->ptrLink;
     Ind = 1;

     while(ptrAux != NULL){
            cout << "Nó: " << Ind << endl;
            cout << "Código do Aluno: " << ptrAux->CodAluno << endl;
            cout << "Nome: " << ptrAux->Nome << endl;
            cout << "Turma: " << ptrAux->Turma << endl;
            cout << endl << endl;

            ptrAux = ptrAux->ptrLink;
            Ind++;
     }
}
```

5.5.6 Desempilhar

A função Desempilhar(...) remove o nó do topo da pilha, caso esta não esteja vazia.

```
bool Desempilhar(DADOS_ALUNO *ptrCab){
     DADOS_ALUNO *ptrAux;
     DADOS_ALUNO *ptrAnterior;
     int Pos;

     //-------------------------------------------------------
     // Se a pilha estiver vazia
     //-------------------------------------------------------
```

```
        if( VerificarPilhaVazia(ptrCab) ){
                cout << "Pilha vazia!";
                return false;
        }

        ptrAux = ptrCab;
        ptrAnterior = ptrCab;

        // Localiza a posição do topo
        while(ptrAux->ptrLink != NULL){
                ptrAnterior = ptrAux;
                ptrAux = ptrAux->ptrLink;
        }

        ptrAnterior->ptrLink = NULL;

        delete ptrAux;
}
```

5.6 Fila dinâmica

Veremos neste tópico a fila simplesmente encadeada, também conhecida como fila dinâmica. A fila dinâmica é um tipo de lista simplesmente encadeada que tem como característica a inserção dos nós inseridos em uma extremidade, e retirados na extremidade oposta. Esta característica, como estudamos antes, é conhecida como FIFO (*First In*, *First Out* - Primeiro a Entrar, Primeiro a Sair) (figura 5.6).

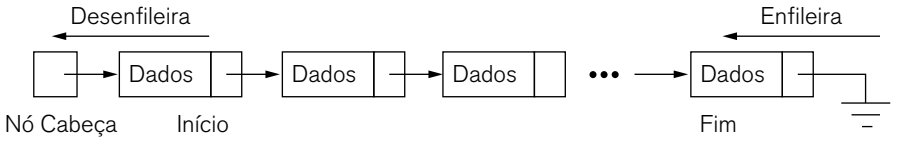


Figura 5.6 – Fila simplesmente encadeada.

5.7 Operações com fila dinâmica.

A seguir serão exibidas as operações básicas com filas.

5.7.1 Criar uma fila vazia

A função CriarFila(...) cria e retorna um primeiro nó. Este primeiro nó, será o nó cabeça.

```
DADOS_ALUNO* CriarFila(){
    DADOS_ALUNO *ptrCab;

    ptrCab = new DADOS_ALUNO;
    ptrCab->ptrLink = NULL;

    return ptrCab;
}
```

5.7.2 Verificar fila vazia

A função a seguir recebe o nó cabeça e verifica se há algum nó na fila.

```
bool VerificarFilaVazia(DADOS_ALUNO *ptrCab){
    if(ptrCab->ptrLink == NULL)
            return true;
    else
            return false;
}
```

5.7.3 Enfileirar

A função Enfileirar(...) insere um novo nó na fila.

```
bool Enfileirar(DADOS_ALUNO *ptrCab, int CodAluno, char Nome[], int
Turma){
```

```
        DADOS_ALUNO *ptrNovo;
        DADOS_ALUNO *ptrAux;
        int Pos;

        //-------------------------------------------------------------
        //                          Cria o novo nó
        //-------------------------------------------------------------
        ptrNovo = new DADOS_ALUNO;
        if (ptrNovo == NULL){
               cout << "Memória insulficiente!";
               return false;
        }

        ptrNovo->CodAluno = CodAluno;
        strcpy(ptrNovo->Nome, Nome);
        ptrNovo->Turma = Turma;
        ptrNovo->ptrLink = NULL;

        //-------------------------------------------------------------
        // Insere no fim da fila
        //-------------------------------------------------------------
        ptrAux = ptrCab;
        while(ptrAux->ptrLink != NULL)
               ptrAux = ptrAux->ptrLink;

        ptrAux->ptrLink = ptrNovo;

        return true;
    }
```

5.7.4 Exibir o primeiro nó da fila

A função ExibirPrimeiro() exibe o primeiro nó da fila.

```
void ExibirPrimeiro(DADOS_ALUNO *ptrCab){
    DADOS_ALUNO *ptrAux;

    ptrAux = ptrCab->ptrLink;


    cout << "Código do Aluno: " << ptrAux->CodAluno << endl;
    cout << "Nome: " << ptrAux->Nome << endl;
    cout << "Turma: " << ptrAux->Turma << endl;
    cout << endl << endl;
}
```

5.7.5 Desenfileirar

A função Desenfileirar(...) remove o nó do início da fila.

```
bool Desenfileirar(DADOS_ALUNO *ptrCab){
    DADOS_ALUNO *ptrAux;
    DADOS_ALUNO *ptrAnterior;
    int Pos;
    //------------------------------------------------------------
    // Se a fila estiver vazia
    //------------------------------------------------------------
    if( VerificarFilaVazia(ptrCab) ){
           cout << "Fila vazia!";
           return false;
    }
    ptrAnterior = ptrCab;
    ptrAux = ptrCab->ptrLink;
    ptrAnterior->ptrLink = ptrAux->ptrLink;
    delete ptrAux;
}
```

5.7.6 Aplicações com Fila

As filas simplesmente encadeadas são úteis em diversas aplicações, como por exemplo, fila de impressão. Os sistemas operacionais, que utilizam filas para gerenciar o escalonamento dos processos que serão executados pelo processador e a alocação de recursos, fila de pedidos de uma expedição ou tele entrega, etc.

5.8 Listas Circulares Simplesmente Encadeadas

As listas circulares diferem das listas encadeadas simples com relação ao seu último nó. Enquanto uma lista encadeada simples tem o seu último nó apontando para um endereço vazio, NULL, uma lista circular aponta para o primeiro. Isto permite que se possa percorrer todos os nós da lista sem acessar uma posição inválida, ou seja, o NULL. Consequentemente, tem que se adotar algum mecanismo que possa identificar quando a lista foi totalmente percorrida. Existem várias técnicas para este fim. Nossa abordagem será utilizar o nó cabeça para reconhecer o início da fila circular.

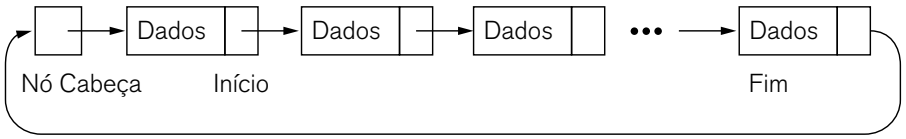


Figura 5.7 - Listas Circulares Simplesmente Encadeadas.

5.9 Operações básicas com listas circulares

Para as operações com listas circulares, a estrutura foi alterada acrescentando a variável membro, `bool NoCabeca`. Esta variável membro se estiver assinalada com `true`, indica que é o nó cabeça, caso contrário é um dos nós da lista. O nó cabeça indicará o início da lista circular.

```
struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
    bool NoCabeca;
    struct DADOS_ALUNO *ptrLink;
};
```

A seguir serão apresentadas as operações básicas com listas circulares simplesmente encadeadas.

5.9.1 Criar lista

A função CriarLista(...) cria e retorna o primeiro nó. Este nó é o único onde teremos ptrCab->NoCabeca = true, indicando que é o nó cabeça.

```
DADOS_ALUNO* CriarLista(){
    DADOS_ALUNO *ptrCab;

    ptrCab = new DADOS_ALUNO;
    ptrCab->NoCabeca = true;
    ptrCab->ptrLink = ptrCab;

    return ptrCab;
}
```

5.9.2 Verificar lista vazia

A função a seguir recebe o nó cabeça e verifica se há algum nó na lista.

```
bool VerificarListaVazia(DADOS_ALUNO *ptrCab){
    if(ptrCab->ptrLink == ptrCab)
            return true;
    else
            return false;
}
```

5.9.3 Inserir um novo nó

A função Inserir(...) insere um novo nó no fim da lista, ou seja, antes do nó cabeça que é o primeiro da lista.

```
   bool Inserir(DADOS_ALUNO *ptrCab, int CodAluno, char Nome[], int
Turma){

        DADOS_ALUNO *ptrNovo;
        DADOS_ALUNO *ptrAux;
        DADOS_ALUNO *ptrAnterior;
        int Pos;

        //-------------------------------------------------------
        //        Cria o novo nó
        //-------------------------------------------------------
        ptrNovo = new DADOS_ALUNO;
        if (ptrNovo == NULL){
               cout << "Memória insulficiente!";
               return false;
        }

        ptrNovo->CodAluno = CodAluno;
        strcpy(ptrNovo->Nome, Nome);
        ptrNovo->Turma = Turma;
        ptrNovo->NoCabeca = false;

        //-------------------------------------------------------
        // Insere no fim da lista
        //-------------------------------------------------------
        ptrAnterior = ptrCab;
        ptrAux = ptrCab->ptrLink;

        while(ptrAux->NoCabeca != true){
               ptrAnterior = ptrAux;
               ptrAux = ptrAux->ptrLink;
```

```
}

    ptrAnterior->ptrLink = ptrNovo;
    ptrNovo->ptrLink = ptrAux;

    return true;
}
```

5.9.4 Exibir lista

A função Exibir(...) exibe todos os nós da lista.

```
void Exibir(DADOS_ALUNO *ptrCab){
    DADOS_ALUNO *ptrAux;

    ptrAux = ptrCab->ptrLink;

    // Conta os nós
    while(ptrAux->NoCabeca != true){
            cout << "Código do Aluno: " << ptrAux->CodAluno << endl;
            cout << "Nome: " << ptrAux->Nome << endl;
            cout << "Turma: " << ptrAux->Turma << endl;
            cout << endl << endl;

            ptrAux = ptrAux->ptrLink;
    }
}
```

5.9.5 Remover nó

A função Remover(...), remove o nó da lista. O nó a ser removido será ao do código do aluno enviado para a função. Uma vez o nó removido, seu espaço é liberado da memória através do comando delete.

```
bool Remover(DADOS_ALUNO *ptrCab, int CodAluno){
    DADOS_ALUNO *ptrAux;
    DADOS_ALUNO *ptrAnterior;
    bool Encontrado;

    //-------------------------------------------------------------
    // Se a lista estiver vazia
    //-------------------------------------------------------------
    if( VerificarListaVazia(ptrCab) ){
           cout << "Lista vazia!";
           return false;
}

    ptrAnterior = ptrCab;
    ptrAux = ptrCab->ptrLink;
    Encontrado = false;

    // Percorre a lista
    while(ptrAux->NoCabeca != true){

           // Se encontrou o código do aluno
           if(ptrAux->CodAluno == CodAluno){
                  Encontrado = true;
                  break;
           }

           ptrAnterior = ptrAux;
           ptrAux = ptrAux->ptrLink;
    }

    if(Encontrado == false){
           cout << "Código do aluno não encontrado!";
           return false;
    }

    ptrAnterior->ptrLink = ptrAux->ptrLink;

    delete ptrAux;
}
```

5.10 Listas Duplamente Encadeadas

Uma deficiência encontrada nas listas circulares simplesmente encadeadas é não poder percorrer os nós em ordem inversa, ou seja, do final para o início. A solução para este tipo de situação são as listas duplamente encadeadas. A estrutura de uma lista duplamente encadeada mantém dois links: um para o próximo nó e um para o nó anterior. Este arranjo permite percorrer a lista em ambas as direções.

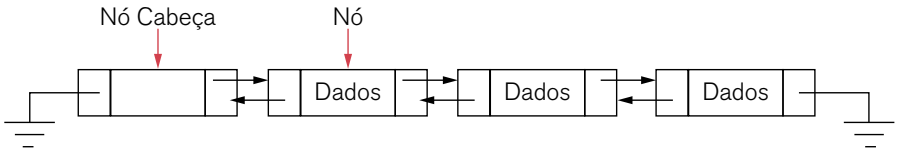


Figura 5.8 - Listas duplamente encadeadas.

CONEXÃO

Leia mais sobre listas duplamente encadeadas em: http://www2.dc.ufscar.br/~bsi/materiais/ed/u10.html

5.11 Operações básicas com listas duplamente encadeadas

Para as operações com listas duplamente encadeadas, a estrutura foi alterada. A estrutura conta com dois ponteiros: ptrLinkProx e ptrLinkAnt. O ponteiro ptrLinkProx aponta para o próximo nó da lista e ptrLinkAnt aponta para o nó anterior.

```
struct DADOS_ALUNO{
    int CodAluno;
    char Nome[100];
    int Turma;
```

```
    struct DADOS_ALUNO *ptrLinkProx;
    struct DADOS_ALUNO *ptrLinkAnt;
};
```

A seguir serão apresentadas as operações básicas com lista simplesmente encadeada.

5.11.1 Criar lista

A função CriarLista(...) cria e retorna o primeiro nó. Este será o nó cabeça.

```
DADOS_ALUNO* CriarLista(){
    DADOS_ALUNO *ptrCab;

    ptrCab = new DADOS_ALUNO;
    ptrCab->ptrLinkProx = NULL;
    ptrCab->ptrLinkAnt = NULL;

    return ptrCab;
}
```

5.11.2 Verificar lista vazia

A função a seguir recebe o nó cabeça e verifica se há algum nó na lista.

```
bool VerificarListaVazia(DADOS_ALUNO *ptrCab){
    if(ptrCab->ptrLinkProx == NULL)
            return true;
    else
            return false;
}
```

5.11.3 Inserir um nó

A função Inserir(...) recebe como parâmetros: o nó cabeça, a posição onde o nó deve ser inserido e os dados referente ao aluno. Inicialmente é criado um nó e atribuído os dados às variáveis membros de sua estrutura. Em seguida, o nó é inserido em uma das seguintes posições: início da lista, em uma posição determinada ou no fim da lista. Se a lista está vazia, o parâmetro PosInserir é ignorado e o novo nó é inserido logo após o nó cabeça. Se é informado alguma valor em PosInserir, então o nó será inserido nesta posição. Caso o valor de PosInserir seja zero, então o nó é inserido no fim da lista.

```
   bool Inserir(DADOS_ALUNO *ptrCab, int PosInserir, int CodAluno,
char Nome[], int Turma){

        DADOS_ALUNO *ptrNovo;
        DADOS_ALUNO *ptrAux;
        DADOS_ALUNO *ptrAnterior;
        int Pos;

        //------------------------------------------------------------
        //Cria o novo nó
        //------------------------------------------------------------
        ptrNovo = new DADOS_ALUNO;
        if (ptrNovo == NULL){
               cout << "Memória insulficiente!";
               return false;
        }

        ptrNovo->CodAluno = CodAluno;
        strcpy(ptrNovo->Nome, Nome);
        ptrNovo->Turma = Turma;
        ptrNovo->ptrLinkProx = NULL;
        ptrNovo->ptrLinkAnt = NULL;
```

```
    //-------------------------------------------------------------
    // Se a lista estiver vazia, insere no início da lista
    //-------------------------------------------------------------
    if( VerificarListaVazia(ptrCab) ){

            // Liga o primeiro nó ao nó cabeça
            ptrCab->ptrLinkProx = ptrNovo;
            ptrNovo->ptrLinkAnt = ptrCab;

            return true;
}
    //-------------------------------------------------------------
    // Se não foi informada a posição, então insere no fim da lista
    //-------------------------------------------------------------
    if (PosInserir == 0)
    {
            // Localiza o último nó
            ptrAux = ptrCab->ptrLinkProx;
            while(ptrAux->ptrLinkProx != NULL)
                    ptrAux = ptrAux->ptrLinkProx;

            ptrAux->ptrLinkProx = ptrNovo;
            ptrNovo->ptrLinkAnt = ptrAux;

            return true;
    }

    //-------------------------------------------------------------
    // Insere na posição informada
    //-------------------------------------------------------------
    ptrAux = ptrCab;
    ptrAnterior = ptrCab;
    Pos = 0;
```

```
        // Localiza a posição a ser inserida
        while((Pos < PosInserir) && (ptrAux != NULL)){
                ptrAnterior = ptrAux;
                ptrAux = ptrAux->ptrLinkProx;
                Pos++;
        }

        if(ptrAux == NULL){
                cout << "Posição não encontrada!";
                return false;
        }

        ptrNovo->ptrLinkProx = ptrAux;
        ptrNovo->ptrLinkAnt = ptrAnterior;
        ptrAnterior->ptrLinkProx = ptrNovo;
        ptrAux->ptrLinkAnt = ptrNovo;
        return true;
}
```

5.11.4 Localizar um nó

A pesquisa na lista utiliza como parâmetro o código do aluno. O código é passado para a função PosicaoNo(...) que retorna a posição do nó, dentro da lista para o código informado.

```
int PosicaoNo(DADOS_ALUNO *ptrCab, int CodAluno){
        DADOS_ALUNO *ptrAux;
        int Posicao;

        //------------------------------------------------------------
        // Se a lista estiver vazia
        //------------------------------------------------------------
        if( VerificarListaVazia(ptrCab) ){
                cout << "Lista vazia!";
                return false;
        }
```

```
        ptrAux = ptrCab;
        Posicao = 0;

        // Localiza o nó
        while(ptrAux != NULL){
                if(ptrAux->CodAluno == CodAluno)
                        break;

                Posicao++;
                ptrAux = ptrAux->ptrLinkProx;
        }


        if(ptrAux == NULL){
                cout << "Código do aluno não encontrado!";
                return -1;
        }

        return Posicao;
}
```

5.11.5 Exibir lista

A função Exibir(...) exibe todos os nós da lista.

```
void Exibir(DADOS_ALUNO *ptrCab){
        DADOS_ALUNO *ptrAux;
        int Ind;

        ptrAux = ptrCab->ptrLinkProx;
        Ind = 1;

        while(ptrAux != NULL){
                cout << "Nó: " << Ind << endl;
                cout << "Código do Aluno: " << ptrAux->CodAluno << endl;
                cout << "Nome: " << ptrAux->Nome << endl;
```

```
            cout << "Turma: " << ptrAux->Turma << endl;
            cout << endl << endl;

            Ind++;
            ptrAux = ptrAux->ptrLinkProx;
        }
    }
```

5.11.6 Remover nó

A função Remover(...), remove o nó da lista em determinada posição. Uma vez o nó removido, seu espaço é liberado da memória através do comando delete.

```
bool Remover(DADOS_ALUNO *ptrCab, int PosRemover){
    DADOS_ALUNO *ptrAux;
    DADOS_ALUNO *ptrAnterior;
    DADOS_ALUNO *ptrProximo;
    int Pos;

    //------------------------------------------------------------
    // Se a lista estiver vazia
    //------------------------------------------------------------
    if( VerificarListaVazia(ptrCab) ){
            cout << "Lista vazia!";
            return false;
    }

    if(PosRemover < 1){
            cout << "Código do aluno não encontrado!";
            return false;
    }

    ptrAux = ptrCab;
    ptrAnterior = ptrCab;
    Pos = 0;
```

```
        // Localiza a posição a ser inserida
        while((Pos < PosRemover) && (ptrAux != NULL)){
                ptrAnterior = ptrAux;
                ptrAux = ptrAux->ptrLinkProx;
                Pos++;
        }


        if(ptrAux == NULL){
                cout << "Código do aluno não encontrado!";
                return false;
        }
        else
        {
                ptrProximo = ptrAux->ptrLinkProx;
        }

        ptrAnterior->ptrLinkProx = ptrProximo;
        if(ptrProximo != NULL)
                ptrProximo->ptrLinkAnt = ptrAnterior;

        delete ptrAux;
}
```

ATIVIDADES

01. De acordo com o material, qual a estrutura de dados que ocupa espaços aleatórios na memória e quando o nó é inserido no fim desta lista, além de armazenar seus os dados, mantém uma ligação com o primeiro nó da lista?

a) Listas Simplesmente Encadeadas.
b) Listas Circulares Simplesmente Encadeadas.
c) Listas Duplamente Encadeadas.
d) Listas Encadeadas Inversamente.
e) Listas Circulares Encadeadas Inversamente.

02. De que forma é implementada uma lista duplamente encadeada para que possa navegar do início para o final da lista e vice versa?

03. Levando em consideração a estrutura veículo abaixo, e que foi criada um ponteiro chamado VeiculoA referente a esta estrutura, qual a forma de atribuir um valor ao ano de fabricação?

```
struct Veiculo{
char Modelo[100];
char Marca [50];
int AnoFabricacao;
int AnoModelo;
};
```

a) VeiculoA->AnoFabricacao = 2015;
b) VeiculoA:AnoFabricacao = 2015;
c) VeiculoA.AnoFabricacao = 2015;
d) VeiculoA&AnoFabricacao = 2015;
e) VeiculoA::AnoFabricacao = 2015;

REFLEXÃO

Neste capítulo estudamos listas lineares encadeadas. Iniciamos nossos estudos implementado as suas operações básicas, para tanto utilizamos os conceitos adquiridos com ponteiros e alocação dinâmica de memória. Continuamos nossos estudos estendendo a implementação para as formas de pilha e filas. Aprofundamos nos aspectos avançados de listas, estudando e implementados as listas circulares simplesmente encadeadas e listas duplamente encadeadas

É importante que, baseado nos conceitos e arquiteturas apresentadas, você seja capaz de identificar as vantagens e desvantagens de cada modelo de estrutura de dados. Reflita sobre isso e identifique as diferenças entre cada uma delas.

Não pare os seus estudos, mantenha-se sempre lendo e pesquisando sobre os diversos temas relacionados as estruturas de dados. Esperamos que todos estes conhecimentos adquiridos sejam um diferencial em sua vida profissional. Desejamos a você um grande sucesso!

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de estruturas de dados e demais assuntos deste capítulo, consulte os capítulos 3 e 4 do livro:

TANENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. Estruturas de Dados Usando C. São Paulo: Makron Books., 1995.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILAR, L. J. **Fundamentos de Programação: Algoritmos, Estruturas de Dados e Objetos.** 3ª ed. São Paulo: McGraw-Hill, 2008.

CELES, W.; RANGEL, J. L. **Apostila de Estruturas de dados.** Disponível em: http://www-usr.inf.ufsm.br/~juvizzotto/elc1067-2013b/estrut-dados-pucrio.pdf. Acesso em: Jan: 2015.

DEITEL, H. M., & DEITEL, P. J. **C++ Como Programar.** 3ª ed. Porto Alegre: Bookman, 2001.

LAUDON, K. C., & LAUDON, J. P. **Sistemas de Informação Gerenciais.** 6ª ed. São Paulo: Prentice Hall, 2007.

MIZRAHI, V. V. **Treinamento em Linguagem C++: Módulos 1 e 2.** 2ª ed. São Paulo: Prentice Hall, 2006.

SCHILDT, H. **C Completo e Total.** 3ª ed. São Paulo: Makron Books Editora Ltda, 1996.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos.** 2ª ed. Rio de Janeiro: LTC, 1994.

TANENBAUM, A. M., LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de Dados Usando C.** São Paulo: Makron Books, 1995.

VELOSO, P. E. **Estruturas de Dados.** 4ª ed. Rio de Janeiro: Campus, 1986.

VILLAS, M. E. **Estruturas de Dados: conceitos e técnicas de implementação.** Rio de Janeiro: Campus, 1993.

GABARITO

Capítulo 1

01. A resposta correta é a D. A pilha é uma estrutura do tipo LIFO (*Last In, First Out* - Último a Entrar, Primeiro a Sair).

02. A diferença entre pilha esta na forma de inserção e remoção dos seus elementos. En-

quanto a pilha é uma estrutura do tipo LIFO (***Last In, First Out*** - Último a Entrar, Primeiro a Sair), a fila é uma estrutura do tipo FIFO (***First In, First Out*** - Primeiro a Entrar, Primeiro a Sair).

03. A resposta certa é grafo. Os grafos são semelhantes às árvores no sentido matemático e são o mais indicado para problemas onde envolve trajetos ou roteamentos.

04. A resposta correta é 3 3 3 3.

05.

```
struct Veiculo{
char Modelo[100];
char Marca [50];
int AnoFabricacao;
int AnoModelo;
};
```

Capítulo 2

01. A resposta certa é a E. Ao inserir um elemento na lista, este é colocado na posição certa de acordo com a ordem pré-estabelecida. Dessa forma, a ordem da lista é mantida e somente a quantidade de elementos é alterada para indicar um elemento a mais na lista.

02. A resposta certa é a D. A pesquisa binária utiliza um algoritmo que efetua divisões sucessivas da lista para encontrar o valor pesquisado.

Capítulo 3

01. As listas lineares sequenciais são estruturas estáticas que têm como objetivo representar um conjunto de dados, que de alguma forma se relacionam, com os elementos dispostos em sequência.

02. As principais vantagens da lista linear sequencial são: qualquer elemento da lista pode ser acessado direto indexado. Tempo constante para acesso a qualquer elemento da lista. Como desvantagens, podemos citar: movimentação de todos os elementos da lista quando há uma inserção ou remoção de elemento. O tamanho máximo da lista deve ser pré-estimado.

03. O nó-cabeça indica o início da lista encadeada e nunca pode ser removido. Dados (registros) do tipo que são armazenados nos demais nós da lista, não deve ser armazenado no nó-cabeça.

Capítulo 4

01. A resposta correta é a B.
02. A resposta certa é a D.
03. A resposta certa é a D.

Capítulo 5

01. A estrutura liga o último nó ao primeiro e a Lista Circular Simplesmente Encadeada. Assim, a opção correta é a B.
02. Para que se possa navegar nos dois sentidos da lista é necessário implementar um algoritmo que ao inserir um nó na lista, além de armazenar os seus dados, deve manter a informação a respeito de quem é seu próximo nó e o anterior.
03. A resposta certa é a A.

ANOTAÇÕES

ANOTAÇÕES