rede e-Tec Brasil

Estrutura de Dados

Nádia Mendes dos Santos
Geraldo Nunes da Silva Júnior
Otílio Paulo da Silva Neto

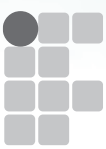
Curso Técnico em Informática




INSTITUTO FEDERAL DE EDUCAÇÃO CIÊNCIA E TECNOLOGIA PIAUÍ
UNIVERSIDADE FEDERAL DE SANTA CATARINA
PDE | PRONATEC
PROGRAMA NACIONAL DE ACESSO AO ENSINO TÉCNICO E EMPREGO
Ministério da Educação

Estrutura de Dados

Nádia Mendes dos Santos
Geraldo Nunes da Silva Júnior
Otílio Paulo da Silva Neto



Teresina - PI
2013

Presidência da República Federativa do Brasil

Ministério da Educação

Secretaria de Educação Profissional e Tecnológica

Este Caderno foi elaborado em parceria entre o Instituto Federal de Educação, Ciência e Tecnologia do Piauí e a Universidade Federal de Santa Catarina para a Rede e-Tec Brasil.

Equipe de Elaboração
Instituto Federal de Educação, Ciência e Tecnologia do Piauí – IFPI

Coordenação Institucional
Francieric Alves de Araujo/IFPI

Coordenação do Curso
Thiago Alves Elias da Silva/IFPI

Professores-autores
Nádia Mendes dos Santos/IFPI
Geraldo Nunes da Silva Júnior/IFPI
Otílio Paulo da Silva Neto/IFPI

Comissão de Acompanhamento e Validação
Universidade Federal de Santa Catarina – UFSC

Coordenação Institucional
Araci Hack Catapan/UFSC

Coordenação do Projeto
Silvia Modesto Nassar/UFSC

Coordenação de Design Instrucional
Beatriz Helena Dal Molin/UNIOESTE e UFSC

Coordenação de Design Gráfico
Juliana Tonietto/UFSC

Design Instrucional
Eleonora Schlemper Mendonça/UFSC
Gustavo Pereira Mateus/UFSC

Web Master
Rafaela Lunardi Comarella/UFSC

Web Design
Beatriz Wilges/UFSC
Mônica Nassar Machuca/UFSC

Diagramação
Bárbara Zardo/UFSC
Roberto Colombo/UFSC

Projeto Gráfico
e-Tec/MEC

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina

S237e Santos, Nádia Mendes dos
Estrutura de dados / Nádia Mendes dos Santos, Geraldo Nunes da Silva Júnior, Otílio Paulo da Silva Neto. – Teresina : Instituto Federal de Educação, Ciência e Tecnologia do Piauí, 2013.
150 p. : il., tabs.

Inclui bibliografia
ISBN: 978-85-67082-02-8

1. Estruturas de dados (Computação). 2. Linguagem de programação(Computadores). 3. Programação (Computadores). I. Silva Júnior, Geraldo Nunes da. II. Silva Neto, Otílio Paulo da. III. Título.

CDU: 681.31:061.6

Apresentação e-Tec Brasil

Bem-vindo a Rede e-Tec Brasil!

Você faz parte de uma rede nacional de ensino, que por sua vez constitui uma das ações do Pronatec - Programa Nacional de Acesso ao Ensino Técnico e Emprego. O Pronatec, instituído pela Lei nº 12.513/2011, tem como objetivo principal expandir, interiorizar e democratizar a oferta de cursos de Educação Profissional e Tecnológica (EPT) para a população brasileira propiciando caminho de o acesso mais rápido ao emprego.

É neste âmbito que as ações da Rede e-Tec Brasil promovem a parceria entre a Secretaria de Educação Profissional e Tecnológica (SETEC) e as instâncias promotoras de ensino técnico como os Institutos Federais, as Secretarias de Educação dos Estados, as Universidades, as Escolas e Colégios Tecnológicos e o Sistema S.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

A Rede e-Tec Brasil leva diversos cursos técnicos a todas as regiões do país, incentivando os estudantes a concluir o ensino médio e realizar uma formação e atualização contínuas. Os cursos são ofertados pelas instituições de educação profissional e o atendimento ao estudante é realizado tanto nas sedes das instituições quanto em suas unidades remotas, os polos.

Os parceiros da Rede e-Tec Brasil acreditam em uma educação profissional qualificada – integradora do ensino médio e educação técnica, - é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação
Janeiro de 2013

Nosso contato
etecbrasil@mec.gov.br

Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.

Atenção: indica pontos de maior relevância no texto.

Saiba mais: oferece novas informações que enriquecem o assunto ou "curiosidades" e notícias recentes relacionadas ao tema estudado.

Glossário: indica a definição de um termo, palavra ou expressão utilizada no texto.

Mídias integradas: sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.

Atividades de aprendizagem: apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.

Sumário

Palavra do professor-autor **9**

Apresentação da disciplina **11**

Projeto instrucional **13**

Aula 1 – Introdução a linguagem de programação C **15**

1.1 Apresentação 15

1.2 Introdução 15

1.3 Estrutura básica de um programa em C 17

1.5 Visão geral: instruções de entrada e saída 17

1.6 Fundamentos em C 18

1.7 Comandos básicos 19

1.8 Tipos de dados 22

1.9 Variáveis 23

1.10 Operadores 25

1.11 Tomada de decisão 28

1.12 Laços 32

Aula 2 – Funções, matrizes, ponteiros e arquivos **37**

2.1 Funções 37

2.2 Matrizes 41

2.3 Ponteiros 43

2.4 Arquivos 46

Aula 3 – Visão geral de estrutura de dados e lista linares **59**

3.1 Introdução 59

3.2 Conceitos básicos 59

Aula 4 – Pilhas **97**

4.1.Introdução 97

4.2.Pilha estática 97

4.3. Pilha dinâmica 102

Aula 5 – Filas e árvores **111**
5.1 Introdução à fila 111
5.2 Fila estática 113
5.3 Fila dinâmica 119
5.4 Introdução à árvore 126
5.5 Árvore binária 128

Aula 6 – Ordenação e pesquisa **135**
6.1 Introdução à ordenação 135
6.2 Método da bolha ou *bubblesort* 136
6.3 Introdução à pesquisa 141
6.4 Pesquisa sequencial 141
6.4 Pesquisa binária 144

Referências **148**

Currículo dos professores-autores **149**

Palavra do professor-autor

Caro estudante!

Bem-vindo à disciplina de Estrutura de Dados. O objetivo desta disciplina é apresentar a Linguagem de Programação C e os seus principais conceitos, bem como uma visão geral acerca de Estrutura de Dados abordando Listas (pilhas, filas) e árvores e uma introdução à ordenação de dados, enfocando o Método da Bolha (*BubbleSort*) e estratégias de pesquisa, especificamente Pesquisa Sequencial e Binária, conceitos, aplicações e implementações, na linguagem C.

Aproveite o curso e a disciplina, pesquise, troque informações com seus colegas e tire dúvidas com seu tutor, pois ele tem muito mais informações para compartilhar com você.

Bons estudos!

Professores
Nádia Mendes dos Santos
Geraldo Nunes da Silva Júnior
Otílio Paulo da Silva Neto

Apresentação da disciplina

As estruturas de dados têm larga aplicação na computação em geral. Sistemas Operacionais e aplicativos as utilizam para várias atividades importantíssimas, como gerenciamento de memória, execução de processos, armazenamento e gerenciamento de dados no disco, etc.

As estruturas de dados definem a organização, métodos de acesso e opções de processamento para a informação manipulada pelo programa. A definição da organização interna de uma estrutura de dados é tarefa do projetista da estrutura, que define também qual a API para a estrutura, ou seja, qual o conjunto de procedimentos que podem ser usados para manipular os dados na estrutura. É esta API que determina a visão funcional da estrutura de dados, que é a única informação relevante para um programador que vá utilizar uma estrutura de dados pré-definida.

Portanto, não faltam motivos para um estudante da área ou qualquer desenvolvedor/programador saberem a fundo e com fluência sobre o assunto.

Projeto instrucional

Disciplina: Estrutura de Dados (carga horária: 90h).

Ementa: Estrutura de um programa em C. Constantes e Variáveis; Operadores, Expressões e Funções; Entrada e Saída. Estruturas de Controle; Funções e Procedimentos. Vetores e Registros; Ponteiros. Estrutura de Dados do Tipo TADs: Listas Lineares Estáticas e Dinâmicas. Pilhas Estáticas e Dinâmicas. Filas Estáticas e Dinâmicas. Árvores: Binárias e Balanceadas. Algoritmos de Percurso. Ordenação e Pesquisa de Dados.

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
1. Introdução à linguagem de programação C	Conhecer e identificar a estrutura básica de um programa em C. Implementar programas básicos na linguagem C. Implementar programas em C utilizando as estruturas de decisão e as estrutura em laços.	Ambiente Virtual de Ensino Aprendizagem (AVEA): http://cefetpi.nucleoead.net/etapi/; Webconferência;	15
2. Funções, matrizes, ponteiros e arquivos	Implementar matrizes, funções na linguagem C. Conceituar e implementar ponteiros. Implementar e classificar os métodos de manipulação de ponteiros. Conceituar e manipular arquivos.	Ambiente Virtual de Ensino Aprendizagem (AVEA): http://cefetpi.nucleoead.net/etapi/; Webconferência;	15
3. Visão geral de estruturas e lista lineares	Conceituar Estrutura de Dados. Descrever os tipos de Estrutura de Dados. Implementar as operações básicas da Estrutura de Dados do tipo Lista estática e dinâmica de ordenação e desordenação.	Ambiente Virtual de Ensino Aprendizagem (AVEA): http://cefetpi.nucleoead.net/etapi/; Webconferência;	15
4. Pilhas	Conhecer o funcionamento de uma Pilha. Implementar as operações básicas em uma Estrutura Pilha Estática e dinâmica.	Ambiente Virtual de Ensino Aprendizagem (AVEA): http://cefetpi.nucleoead.net/etapi/; Webconferência;	15
5. Filas e árvores	Conhecer e identificar uma Fila Estática e Dinâmica. Implementar Fila Estática e Dinâmica na linguagem C. Conceituar Árvore e Árvore Binária; Reconhecer os percursos em árvores binárias.	Ambiente Virtual de Ensino Aprendizagem (AVEA): http://cefetpi.nucleoead.net/etapi/; Webconferência;	15
6. Ordenação e pesquisa	Classificar ou ordenar dados. Implementar o método da Bolha ou *BubbleSort*. Conceituar e implementar diferentes estratégias de pesquisa.	Ambiente Virtual de Ensino Aprendizagem (AVEA): http://cefetpi.nucleoead.net/etapi/; Webconferência;	15

Aula 1 – Introdução a linguagem de programação C

Objetivos

Conhecer e identificar a estrutura básica de um programa em C.

Implementar programas básicos na linguagem C.

Implementar programas em C utilizando as estruturas de decisão e as estrutura em laços.

1.1 Apresentação

Habitualmente antes de resolvermos exemplos ou exercícios, elaboraremos o algoritmo, que nada mais é que uma sequência de operações cuja execução produz um resultado que é a resposta de um problema proposto.

Um programa de computador nada mais é que a codificação de um algoritmo numa linguagem de programação. Linguagens como C, *Pascal*, BASIC, ALGOL, *Clipper*, COBOL, etc., são chamadas procedurais, devido ao fato das instruções serem executadas de forma sequencial, enquanto que as linguagens baseadas no conceito de eventos como C++, *Visual BASIC*, *Visual Objects*, utilizam outra estratégia de programação (Programação Orientada ao Objeto), em C utilizaremos a metodologia estruturada.

1.2 Introdução

A linguagem C foi desenvolvida inicialmente por Dennis M. Ritchie e Ken Tompson no laboratório *Bell* no ano de 1972. Baseada na linguagem B criada por Tompson, esta linguagem evoluiu da linguagem BCPL, dando origem as duas linguagens anteriores.

C foi inicialmente projetada para ser utilizada no sistema operacional *Unix*. Podemos definir a linguagem C como sendo uma linguagem de programação robusta e multiplataforma, projetada para aplicações modulares de rápido acesso.

A-Z

ANSI
É a sigla para American National Standards Institute e designa uma organização americana que tem a função de estabelecer quais normas desenvolvidas devem virar padrão.

Veremos aqui a estrutura **ANSI** C, versão padrão definida pelo comitê americano ANSI e suportada praticamente por todos os compiladores C.

Podendo ser considerada como uma linguagem de médio nível, pois possui instruções que a tornam ora uma linguagem de alto nível e estruturada como o Pascal, se assim se fizer necessário, ora uma linguagem de baixo nível, pois possui instruções tão próximas da máquina, que só o *Assembler* possui.

De fato com a linguagem C podemos construir programas organizados e concisos (como o *Pascal*), ocupando pouco espaço de memória com alta velocidade de execução (como o *Assembler*). Infelizmente, dada toda a flexibilidade da linguagem, também poderemos escrever programas desorganizados e difíceis de serem compreendidos (como usualmente são os programas em BASIC).

Devemos lembrar que a linguagem C foi desenvolvida a partir da necessidade de se escrever programas que utilizassem recursos próprios da linguagem de máquina de uma forma mais simples e portável que o *Assembler*.

Linguagem C é *case sensitive*, ou seja, diferencia letras maiúsculas de minúsculas.

As inúmeras razões para a escolha da linguagem C como a predileta para os desenvolvedores "profissionais". As características da Linguagem C servirão para mostrar o porquê de sua ampla utilização.

1.2.1 Características da linguagem C

- Portabilidade entre máquinas e sistemas operacionais.
- Dados compostos em forma estruturada.
- Programas Estruturados.
- Total interação com o Sistema Operacional.
- Código compacto e rápido, quando comparado ao código de outras linguagem de complexidade análoga.

1.3 Estrutura básica de um programa em C

Um programa em C consistem em uma ou várias "funções". Vamos começar pelo menor programa possível em C:

```
main ( ) {

}
```

Todo programa em C tem necessariamente a função *main*.

Este programa compõe-se de uma única função chamada *main*.

Instruções de programa

Vamos adicionar uma instrução ao nosso programa.

```
main ( ){

printf("olá"); /* mostra na tela a mensagem Olá*/

}
```

1.4.1 Comentários

Os comentários servem principalmente para documentação do programa e são ignorados pelo compilador, portanto não irão afetar o programa executável gerado. Os comentários iniciam com o símbolo **/*** e se estendem até aparecer o símbolo ***/**. Um comentário pode aparecer em qualquer lugar no programa onde possa aparecer um espaço em branco e pode se estender por mais de uma linha.

1.5 Visão geral: instruções de entrada e saída

Toda linguagem de programação de alto nível suporta o conceito de "Tipo de dado", que define um conjunto de valores que a variável pode armazenar, e os tipos mais comuns encontrados nas linguagens de programação, ou seja, inteiro, real e caractere. Diferentemente do *Pascal* que é fortemente tipada onde a mistura entre um número inteiro e um real pode causar erros, C suporta livremente tipos caracteres e inteiros na maioria das expressões!

Por ser capaz de manipular *bits*, *bytes* e endereços, C se adapta bem a programação em nível de sistema. E tudo isto é realizado por apenas 43 palavras reservadas no Turbo C, 32 nos compiladores padrão *ANSI* e 28 no C Padrão. Como curiosidade, o *IBM BASIC* que é um interpretador *BASIC* com fins puramente educativos tem 159 comandos.

1.6 Fundamentos em C

Desta forma a orientação que adotaremos neste início do curso se deterá mais na compreensão geral do programa, do que a análise detalhada de cada comando ou função utilizada. De fato apresentaremos alguns comandos fundamentais para a escrita de programas básicos e apenas nos utilizaremos sua sintaxe mais elementar (posteriormente estudaremos cada um deles mais detidamente), construiremos os primeiros programas do curso.

Exemplo 1: Programa mostra a idade.

```
/* Exemplo Idade */

main ( ) {

int idade;

idade = 40;

printf("Sua idade e' %d anos. \n", idade);

}
```

Este programa simplesmente imprime "Sua idade e' 40 anos." saltando uma linha **(/n)** em seu término.

1.6.1 Diretiva *#include*

Toda a diretiva, em C, começa com o símbolo **#** no início da linha. A diretiva *#include* inclui o conteúdo de outro arquivo dentro do programa atual, ou seja, a linha que contêm a diretiva é substituída pelo conteúdo do arquivo especificado.

Sintaxe:

```
#include <nome do arquivo>

ou

#include "nome do arquivo"
```

O primeiro caso é o mais utilizado. Ele serve para incluir alguns arquivos que contêm declaração das funções da biblioteca padrão, entre outras coisas. Estes arquivos, normalmente, possuem a extensão .h e se encontram em algum diretório pré-definido pelo compilador (*/usr/include no Linux; c:\dev-c++\include* no *Windows* com o *Dev-C++*). Sempre que o programa utilizar alguma função da biblioteca-padrão deve ser incluído o arquivo correspondente. A tabela a seguir apresenta alguns dos principais **. h** da linguagem C:

Arquivo Descrição -

- *stdio.h* ->Funções de entrada e saída (I/O)
- *string.h* ->Funções de tratamento de strings
- *math.h* ->Funções matemáticas
- *ctype.h* ->Funções de teste e tratamento de caracteres
- *stdlib.h* ->Funções de uso genérico

A segunda forma, onde o nome do arquivo aparece entre aspas duplas, serve normalmente para incluir algum arquivo que tenha sido criado pelo próprio programador ou por terceiros e que se encontre no diretório atual, ou seja, no mesmo diretório do programa que está sendo compilado.

1.7 Comandos básicos

As instruções de entrada e saída são comandos quase que obrigatórios em qualquer programa na linguagem C. Na maioria dos programas em C, o usuário necessita entrar com algumas informações e também saber o resultado de algum processamento. Descreve-se a seguir os comandos básicos da linguagem C.

1.7.1 Instruções de entrada e saída

O objetivo de escrevermos programas é em última análise, a obtenção de resultados (Saídas) depois da elaboração de cálculos ou pesquisas (Processamento) através do fornecimento de um conjunto de dados ou informações conhecidas (Entradas).

1.7.2 A função *printf ( )*

É um dos mais poderosos recursos da linguagem C, printf( ) servirá basicamente para a apresentação de dados no monitor.

Sua forma geral será: *printf* ("*string* de controle", lista de argumentos);

Necessariamente você precisará ter tantos argumentos quantos forem os comandos de formatação na "*string* de controle". Se isto não ocorrer, a tela exibirá sujeira ou não exibirá qualquer dado.

Exemplo: Dado um número, calcule seu quadrado.

```
#include <stdio.h>
#include <conio.h>
main ( ) {
int numero;
numero=10;
printf("O %d elevado ao quadrado resulta em %d. \n",
numero,numero*numero);
        getch ( );
}
```

A diretiva *#include* foi utilizada, pois usamos o comando *printf (stdio.h)* e o comando *getch (conio.h).*

Observemos o Quadro de Operadores Especiais suportados por *printf( )*:

Código	Significado
\b	Retrocesso (*BackSpace*)
\f	Salto de Página (*Form Feed*)
\n	Linha Nova (*Line Feed*)
\t	Tabulação Horizontal (*TAB*)
\x	Representação de *byte* na base hexadecimal

Exemplo: *printf("\x41")*; causa a impressão da letra A na tela.

1.7.3 A função *scanf ( )*

Uma das mais importantes e poderosas instruções, servirá basicamente para promover leitura de dados (tipados) via teclado.

Sua forma geral será: *scanf("string de controle", lista de argumentos);*

Posteriormente ao vermos sua sintaxe completa, abordaremos os recursos mais poderosos da <*string* de controle>, no momento bastará saber que:

%c - leitura de *caracter*;

%d - leitura de números inteiros;

%f - leitura de números reais;

%s - leitura de caracteres.

A lista de argumentos deve conter exatamente o mesmo número de argumentos quantos forem os códigos de formatação na <*string* de controle>. Se este não for o caso, diversos problemas poderão ocorrer - incluindo até mesmo a queda do sistema - quando estivermos utilizando programas compilados escritos em C.

Cada variável a ser lida, deverá ser precedida pelo *caracter &*, por razões que no momento não convém explicarmos, mas que serão esclarecidas no decorrer do curso. Para sequência de caracteres (%s), o *caracter &* não deverá ser usado.

Exemplo: Programa para ler e mostrar uma idade.

```
/* Exemplo Lê e Mostra Idade */
main ( ) {
int idade;
char nome[30];
printf("Digite sua Idade: ");
scanf("%d",&idade);
printf("Seu Nome: ");
```

```
scanf("%s",nome); /* Strings não utilizar '&' na leitura */
printf("%s sua idade e' %d anos. \n", nome, idade);
}
```

Faça um programa em C para perguntar sua profissão e mostrar na tela. Poste no AVEA da disciplina o arquivo de sua atividade.

1.8 Tipos de dados

No momento dispomos de conhecimento para elaboração de programas básicos para construção de pequenos programas, pois conhecemos instruções de entrada de dados (*scanf*) e de saída *(printf)*.

Veremos a seguir Tipos de Dados da linguagem C, variáveis, operadores, e demais instruções básicas. Devemos procurar compreender a utilidade das declarações que serão exibidas a seguir, relacionando estes recursos com os exemplos e exercícios vistos anteriormente.

Semelhante ao *BASIC*, *Pascal* e *COBOL*, a linguagem C necessita que todas as variáveis tenham seus tipos definidos. C aceita tipos básicos (caractere, inteiro, ponto flutuante, dupla precisão e sem valor) e modificadores (sinal, sem sinal, longo e curto) que podem alterar os tipos básicos. Observemos o Quadro 1.1 para melhor entendimento.

Quadro 1.1: Tabela de tamanhos e escala de tipos básicos

Tipo	Extensão	Escala Numérica em bits
Char	8	0 a 255
Int	16	-32768 a 32767
Float	32	3.4E-38 a 3.4E+38
Double	64	1.7E-308 a 1.7E+308
Void	0	Sem valor

Fonte: Schildt H, 1997

Exemplo: Mesmo número com duas representações diferentes.

```
main ( ) {
float a;
printf("Digite um numero: ");
```

```
scanf(“%f”,&a);

printf(“%f %e”,a,a);

}
```

Simulando obtemos:

Digite um número: 65
65.000000 6.500000E+01

1.9 Variáveis

As variáveis tem uma importância fundamental na linguagem C. São utilizadas como depósitos temporários de informações a serem processados pelo programa escrito na linguagem C.

1.9.1 Variáveis

De maneira semelhante as demais linguagens estruturadas, em C as variáveis tem seus valores localizados nas rotinas onde foram declaradas (escopo).

Todas as variáveis devem ser declaradas desta forma:

“tipo nome_de_variaveis;”

Exemplos:

```
int i,j,l;

short int si;

double balanco, consolidacao;

char nome[30];
```

Basicamente, as variáveis podem ser declaradas fora das funções (globais) que valem para todas as funções do programa. Podem ser declaradas dentro de uma função (locais) sendo desconhecida no restante do programa. Além disso podem ser usadas para passagem de valores entre funções (parâmetros).

1.9.2 Inicializando variáveis

Em C podemos criar uma variável e logo em seguida utilizarmos o operador de atribuição para inicializarmos o valor de uma variável. Este processo denomina-se inicialização de variáveis. Vejamos abaixo como fazemos tal procedimento.

```
int i=0;

double x=10.5;

Exemplo: Criando três variáveis e inicializando-as em tempo de criação.

main ( ) {

int evento = 5;

char corrida = 'c';

float tempo = 27.25;

printf (" o melhor tempo da eliminatória % c", corrida);

printf (" \ n do evento %d foi % f", evento, tempo);

}
```

Simulando obtemos:

o melhor tempo da eliminatória c
do evento 5 foi 27.25

1.9.3 Nomes de variáveis

A escolha de nomes de variáveis em C pode conter quantos caracteres quiser, sendo o primeiro caractere obrigatoriamente uma letra ou o caractere sublinhado. Outra característica marcante em C, é que os nomes de variáveis podem ser criadas com nomes iguais, desde que contenham letras maiúsculas ou minúsculas, para diferenciá-las. É isso mesmo, C fazer distinção entre letras maiúsculas e minúsculas.

```
int valor;

double Valor;
```

Observem que as variáveis possuem os mesmos nomes, só que a primeiro é denominada valor escrita em minúsculo e a segunda denominada Valor, sendo o primeiro caractere escrito em maiúscula.

1.10 Operadores

C é uma das linguagens com maior número de operadores, devido possuir todos os operadores comuns de uma linguagem de alto nível, porém também possuindo os operadores mais usuais a linguagens de baixo nível. Para fins didáticos, dividiremos os operadores em aritméticos, lógicos e de *bits*. No momento abordaremos apenas as duas primeiras classes.

1.10.1 Operadores aritméticos

Os operadores aritméticos são utilizados para efetuar as operações matemáticas básicas como adição, multiplicação, divisão, resto de divisão inteira, subtração o menos unário, decremento e incremento. O Quadro 1.2 estabelece o operador aritmético e a ação de cada um.

Quadro 1.2: Operadores Aritméticos

Operador	Ação
+	Adição
*	Multiplicação
/	Divisão
%	Resto de Divisão Inteira
-	Subtração o menos unário
--	Decremento
++	Incremento

Fonte: Schildt H, 1997

1.10.2 Operadores relacionais e lógicos

Os operadores relacionais e lógicos são utilizados para efetuar as operações matemáticas relacionais e lógicos que são os seguintes: como maior que, maior ou igual que, menor que, menor ou igual que, igual a, diferente de condição e. O Quadro 1.3 estabelece os operadores relacionais e lógicos e a ação de cada um.

Quadro 1.3: Operadores relacionais e lógicos

Operador	Ação
>	Maior que
>=	Maior ou igual que
<	Menor que
<=	Menor ou igual que
==	Igual a
!=	Diferente de
&&	Condição "E"

Fonte: Schildt H, 1997

Em C o resultado da comparação será ZERO se resultar em FALSO e DIFERENTE DE ZERO no caso de obtermos VERDADEIRO num teste qualquer. Programadores experientes utilizam-se desta conclusão em alguns programas, onde inexplicavelmente algo é testado contra ZERO.

Exemplo: Irá calcular a média de um aluno sendo que ele possua três notas, a média é a soma das notas dividida por três.

```
#include <stdio.h>
#include <conio.h>
void main ( )
{
clrscr ( );
float nota1;
float nota2;
float nota3;
float media;
printf("\n Digite a primeira nota..: ");
scanf("%f",&nota1);
printf("\n Digite a segunda nota...: ");
scanf("%f",&nota2);
printf("\n Digite a terceira nota..: ");
scanf("%f",&nota3);
media=(nota1+nota2+nota3)/3;
printf("\n\n Sua média .....:%6.2f",media);
getch ( );
}
```

Faça um programa em C onde dado 5 números, imprima-os em ordem crescente. Poste no AVEA da disciplina o arquivo de sua atividade.

1.10.3 Incremento e decremento

Para efetuar o incremento de uma variável deste tipo, é só acrescentarmos (++) seguido do nome da variável e para decrementar utilizamos (--) seguido do nome da variável. Outra maneira de usarmos o incremento e decremento é colocar o (++) ou (--) antecedendo o nome da variável.

Na ocorrência de (++) ou (--) após o nome da variável, denominados pós-fixado, e antecedendo o nome da variável, denominados prefixado. Vejamos alguns exemplos:

```
n = n + 1; /* adiciona 1 a n */


//seria o mesmo que

++n; /* adiciona 1 a n */


//ou

n++; /* adiciona 1 a n */
```

O que muda nas operações a cima, é apenas o uso dos incrementos, pós--fixados e prefixados, mas que fazem bastante diferenças quando utilizados em conjunto com outras instruções.

Vejamos o exemplo:

```
n = 10;

m = ++n

printf (“\n N=%d M=%d”, n, m);

a = 10

x = a++

printf(“\n A=%d X=%d”, a, x);
```

Simulando obtemos:

N=10 M=11

A=10 X=10

1.11 Tomada de decisão

A tomada de decisão e um dos procedimentos mais comuns em programas de computadores, pois é através deles que o computador decide que procedimento executar. A linguagem C nos permite utilizar três comando de decisão:

- *if*
- *if-else*
- *switch*

1.11.1 Comando *if-else*

Análogo a outras linguagens, sua forma geral será:

```
if <condição>
        <comando>;
else
        <comando>;
```

Exemplo 1: Programa Adulto, Jovem ou Velho.

```
main ( ) {
int i;
printf("Digite sua idade: ");
scanf("%d",&i);
if (i > 70)
                printf("Esta Velho!");
else
                if (i > 21)
```

```
                    printf("Adulto");
            else
                    printf("Jovem");
}
```

A expressão avaliada deverá obrigatoriamente estar entre parênteses.

Exemplo 2: Maior entre três números.

```
main ( ) {
int a,b,c;
clrscr();
printf("Digite o 1º Número: ");
scanf("%d",&a);
printf("\nDigite o 2º Número: ");
scanf("%d",&b);
printf("\nDigite o 3º Número: ");
scanf("%d",&c);
if (a > b)
            if (a > c)
                    printf("\nO Maior é %d",a);
            else
                    printf("\nO Maior é %d",c);
else
            if (b > c)
                    printf("\nO Maior é %d",b);
            else
                    printf("\nO Maior é %d",c);
}
```

Exemplo 3: Maior entre três números (segunda solução).

```
main ( ) {
int a,b,c,d;
clrscr();
printf("Digite o 1º Número: ");
scanf("%d",&a);
printf("\nDigite o 2º Número: ");
scanf("%d",&b);
printf("\nDigite o 3º Número: ");
scanf("%d",&c);
if (a > b)
                    d = a;
else
                    d = b;
if (c > d)
                    printf("\nO Maior é %d",c);
else
                    printf("\nO Maior é %d",d);
}
```

1.11.2 Comando *switch*

Este comando nos permite selecionar uma opção entre várias alternativas. A variável será validada e conforme seu valor executará a opção na qual se enquadra. Sua forma geral será:

```
switch (opçãol) {
        case <valor1>: instrução;
                                break;
        case <valor2>: instrução;
                                break;
        case <valor3>: instrução;
                                break;
        default : instrução;
}
```

Exemplo 1: Programa adulto ou velho.

```
#include <stdio.h>
#include <stdlib.h>
int main ( )
{
int i;
char sn;
printf("Voce tem mais de 70 anos ? (S/N) ");
scanf("%c",&sn);
switch (sn) {
case 's' : printf("Voce Esta Velho! \n"); break;
case 'S' : printf("Voce Esta Velho! \n"); break;
case 'n' : printf("Voce Esta Adulto! \n"); break;
case 'N' : printf("Voce Esta Velho! \n"); break;
}
system ("pause");
}
```

1.12 Laços

Um dos grandes benefícios dos sistemas de processamento de dados está em sua confiabilidade (precisão nos cálculos) e rapidez (infinitamente superior ao ser humano), desta forma é ideal para processamento de elevado número de operações repetitivas. O processamento de uma Folha de Pagamentos, a Emissão de Notas Fiscais, a Geração de Estatísticas de Faturamento, são típicas tarefas a serem realizadas por processamento eletrônico de dados.

1.12.1 Comando *for*

O laço for é a instrução mais poderosa na criação de estruturas de repetição. Neste momento, abordaremos apenas sua sintaxe simplificada, Sua forma mais simples é:

for (<início>;<condição>;<incremento>) comando;

Exemplo 1: Contagem de 1 a 100 ficaria.

```
main ( ) {

int cont;

for (cont = 1; cont <= 100; cont++)

                printf("%d",cont);

}
```

Exemplo 2: Elaborar programa que imprima a tabuada de um número dado.

```
main ( ) {

int cont,num;

printf("Digite um Numero: "); scanf("%d",&num);

for (cont = 0; cont <= 10; cont++)

                printf("%2d * %2d = %2d \n",num,cont,num * cont);

}
```

O número ‘2’ antes do ‘d’ causa a representação em vídeo de 2 casas, permitindo o alinhamento da tabuada!.

Exercício1: Elabore tabela de Conversão de temperaturas entre as escalas *Celsius* e *Fahrnheit*.

```
#include <stdio.h>

#include <stdlib.h>

main ( ) {

int fahr;

float celsius;

for (fahr = 0; fahr <= 300; fahr = fahr + 20) {

                    printf(“%4d”, fahr);

                    celsius = (5.0/9.0)*(fahr-32);

                    printf(“\t%6.1f\n”,celsius);

}

}
```

Quando dois ou mais comandos devam ser executados, devemos obrigatoriamente utilizar chaves para delimitar a sequência de instruções a ser observada.

1.12.2 Comando *while*

O laço while significa enquanto, e é um comando utilizado para se realizar repetições quando não se pode determinar a quantidade de vezes que será repetido o laço. Diferente do for que se determina qual a quantidade de vezes o laço se repete. Neste momento, abordaremos apenas sua sintaxe simplificada, Sua forma mais simples é:

```
while (<condição>) {

        <instrução>;

}
```

Exemplo 1: Contagem de 1 a 100 ficaria.

```
main ( ) {

int cont=0;

while (cont = 100)

                cont++;

        printf(“%d”,cont);

}
```

1.12.3 Comando *do-while*

O laço do-while significa faça enquanto, e é um comando utilizado para se realizar repetições com um diferencial que é a execução de pelo menos uma vez a instrução do laço, mesmo que a condição inicial seja falsa, pois a validação da condição só é feita depois da execução do laço. Neste momento, abordaremos apenas sua sintaxe simplificada, Sua forma mais simples é:

```
do {

        <instrução>;

} while (<condição>);
```

Exemplo 1: Contagem de 1 a 100 ficaria:

```
#include <stdio.h>

#include <stdlib.h>



int main() {

int cont=0;

do {

                    cont++;

                    printf(" %d ",cont);

} while (cont < 100);

system("pause");

}
```

Assista à video-aula da disciplina de Estrutura de Dados disponível em http://cefetpi.nucleoead.net/etapi/. Aproveite para revisar os conteúdos da aula sobre Introdução a Linguagem de Programação C. Depois disto, anote em um documento Word suas dúvidas e poste estas anotações no fórum do AVEA, para que você compartilhe dúvidas e sane dificuldades.

Resumo

Nesta aula abordamos conceitos da linguagem de programação C, a sua estrutura básica de um programação, suas características, vantagens, desvantagens, operações, declarações e manipulações de variáveis e as implementações das estruturas condicionais, bem como as implementações de programas das estruturas de laço. Além de exemplos práticos do nosso dia-a-dia, a fim de um entendimento melhor por parte do leitor aumentando assim seu conhecimento teórico.

Atividades de aprendizagem

1. Elabore um programa em C que faça o seguinte:

a) Mostre na tela o seu nome.

b) Some 10 com 15 e imprima na tela a seguinte frase: "O resultado da soma é: " mostrando o resultado.

c) Faça a multiplicação entre 10 e 15 e mostre na tela o resultado.

d) Pergunte o seu nome e imprima na tela.

e) Pergunte o nome da pessoa e o sobrenome e imprimir os mesmos.

f) Leia dois números e apresente seu produto.

g) Leia dois números e apresenta a sua subtração.

h) Leia o nome e as duas notas de um aluno e apresente ambos na tela.

Aula 2 – Funções, matrizes, ponteiros e arquivos

Objetivos

Implementar matrizes, funções na linguagem C.

Conceituar e implementar ponteiros.

Implementar e classificar os métodos de manipulação de ponteiros.

Conceituar e manipular arquivos.

2.1 Funções

Conceitualmente, C é baseada em blocos de construção. Assim sendo, um programa em C nada mais é que um conjunto de funções básicas ordenadas pelo programador. As instruções *printf( )* e *scanf*( ), vistas anteriormente, não fazem parte do conjunto de palavras padrões da linguagem (instruções), pois não passam elas mesmas de funções escritas em C! Esta abordagem permite a portabilidade da linguagem, pois seus comandos de entaada e saída, não são parte do conjunto básico da linguagem, livrando-a desta forma dos problemas de suporte aos diversos padrões de vídeos, teclados e sistemas operacionais existentes.

Cada função C é na verdade uma sub-rotina que contém um ou mais comandos em C e que executa uma ou mais tarefas. Em um programa bem escrito, cada função deve executar uma tarefa. Esta função deverá possuir um nome e a lista de argumentos que receberá. As funções em C são muito semelhantes às usadas no Pascal, com a diferença que o próprio programa principal é apenas uma função que se inicia com a palavra reservada *main ( )* podendo receber parâmetros diretamente do *DOS*, por exemplo.

Exemplo: Programa principal chamando função alo.

```
main ( ) {
                    alo ( );
}
```

```
alo ( ) {
                printf (“Alô!\n\n”);
}
```

Retomemos o exemplo do cálculo de um número elevado ao quadrado.

Exemplo: Quadrado com função.

```
main ( ) {
int num;
printf(“Digite um numero: “);
scanf(“%d”,&num);
sqr(num); /* sqr recebe “num” do programa principal */
}
sqr ( ) {
                int x; /* x é um “parâmetro” recebido do programa principal
                no caso x “vale” o conteúdo de num */
                printf(“%d ao quadrado e’ %d “,x,x*x);
}
```

O argumento simplesmente é o valor (em “num”) digitado no programa principal (em *scanf*) e enviado a função *sqr*.

Um conceito importante e normalmente confundido é a diferença conceitual entre “argumento” e “parâmetro” que em resumo pode ser definido da seguinte forma: “Argumento” se refere ao valor que é usado para chamar uma função. O termo “Parâmetro” se refere à variável em uma função que recebe o valor dos argumentos usados na função. A distinção que deve ser compreendida é que a variável usada como argumento na chamada de uma função não tem nenhuma relação com o parâmetro formal que recebe o valor dessa variável.

Exercício: Passagem de variáveis entre rotinas.

```
int x;
main ( ) {
int a;
printf("Digite um valor: ");
scanf("%d",&a);
x = 2 * a + 3;
printf("%d e %d",x,soma(a));
}
soma (z) {
                int z;
                x = 2 * x + z;
                return(x);
}
```

2.1.1 Protótipo de uma função

A declaração de uma função quando feita no inicio de um programa em C é dita protótipo da função. Esta declaração deve ser feita sempre antes da função *main*, definindo-se o tipo, o nome e os argumentos desta mesma função. Exemplo:

```
float soma (float, float);
```

Definindo-se o protótipo de uma função não é necessário escrever o código desta mesma função antes da função *main*, pois o protótipo indica ao compilador C que a função está definida em outro local do código. Vejamos agora a utilização do exemplo anterior usando o protótipo de uma função.

```
#include <stdio.h>
#include <stdlib.h>
int sqr (int); /* protótipo da função*/
int main ( ) {
 int num;
 printf("Digite um numero: ");
 scanf("%d",&num);
 sqr(num); /* sqr recebe "num" do programa principal */
 system("pause");
}
int sqr(int num) {
 int x = num; /* x é um "parâmetro" recebido do programa principal
 no caso x "vale" o conteúdo de num */
 printf("%d ao quadrado e' %d \n",x,x*x);
}
```

2.1.2 Função recursiva

Uma função denomina-se recursiva quando dentro dela se faz uma chamada para ela mesma. Um exemplo prático seria o cálculo do fatorial de um número.

Faça uma função que escreva o endereço da variável ZEND. Poste no AVEA da disciplina o arquivo de sua atividade.

```
#include <stdio.h>
#include <stdlib.h>
long fatorial (int);
int main() {
 int n;
 do {
```

```
printf("Digite um numero ou negativo p/ sair \n");
 scanf("%d",&n);
 if (n < 0) {
            break;
            }
 printf("O Fatorial de %d eh %d \n",n,fatorial(n));
 } while (1);
 system("pause");
 return 0;
}
long fatorial (int n) {
 return ((n==0) ? (long)1 : (long)n * fatorial(n-1) );
}
```

2.2 Matrizes

Uma Matriz é um conjunto de variáveis de mesmo tipo que compartilham um mesmo nome. Com Matriz agora podemos armazenar mais de um valor para depois serem manipulados através de um índice, o qual referencia cada um dos elementos, para se criar uma Matriz é necessário definir um tipo, seu nome e quantidade de elementos, sedo este ultimo entre colchetes ([ ]). Vejamos agora a declaração de uma matriz.

```
int mat[5];
```

2.2.1 Referenciação e Atribuição dos elementos de uma matriz

Para se referenciar um elemento da Matriz individualmente, basta apenas colocar-se o nome desta Matriz seguida do seu índice entre colchetes.

Os índices dos elementos de uma matriz são sempre iniciados por zero.

Vejamos um exemplo para se referenciar um elemento da Matriz:

```
x = mat[10]; /* x recebe o elemento de mat na posição 10 */

mat[10] = 20; /* o elemento de mat na posição 10 recebe o valor 20 */
```

2.2.2 Inicialização de matrizes

Para se inicializar uma Matriz é necessário apenas colocar-se o nome desta Matriz, a quantidade elementos entre colchetes seguida do operador de atribuição e por fim os valores separados por ponto e vírgula entre as chaves ({ }). Vejamos um exemplo para se iniciar uma Matriz.

```
int mat[3] = {1;2;3}; /* matriz inicializada com os valores 1,2 e 3 */
```

Para exemplificar melhor, vejamos o exemplo abaixo de um programa que calcula a media de três notas.

```
#include <stdio.h>

#include <stdlib.h>

int main() {

 int i;

 float nota[3], m=0;

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

 printf("\n Digite a nota %d ", i+1 );

 scanf("%f",&nota[i]);

 m = m + nota[i];

 }

 m /= 3;

 printf("\n A media e' %.2f \n", m );

 system("pause");

 return 0;

}
```

2.3 Ponteiros

O ponteiro nada mais é do que uma variável que guarda o endereço de outra variável.

2.3.1 Declaração de ponteiros

A declaração de ponteiros é feita da seguinte forma:

```
int *ptr
```

Conforme qualquer variável em C, o ponteiro deve ser declarado antes de ser usado. Basta inserir o operador indireto (*) após o tipo da variável:

```
int *ptr;
```

Essa declaração define a variável *ptr* como um ponteiro para uma variável do tipo *int* (número inteiro).

A declaração de ponteiro não tem o mesmo significado da declaração de uma variável. Ela indica apenas o tipo de objeto de dados apontado e, desde que contém endereço de memória, o tamanho em *bytes* que ocupa não tem relação com o tamanho do objeto apontado. O tamanho do ponteiro é fixo e depende apenas do modelo de memória do sistema (2 *bytes* ou 4 *bytes*, normalmente).

Para declarar mais de um ponteiro por linha, usa-se um operador indireto (*) para cada

```
char *ch1, *ch2; (são ponteiros para o tipo char).
```

Se um operador for omitido (exemplo: *char *ch1, ch2;*), a variável correspondente não será ponteiro e, certamente, provocará erro de execução se usada como tal.

O ponteiro pode ser declarado para qualquer tipo legal de variável em C (*char, int, float, double,* etc), além de void, que seria um genérico, podendo apontar para qualquer tipo de dado.

2.3.2 Inicialização de ponteiros

Para se inicializar um ponteiro é necessário apenas atribuir-se um endereço de memória.

A simples declaração de um ponteiro não o faz útil. É necessária a indicação da variável para a qual ele aponta.

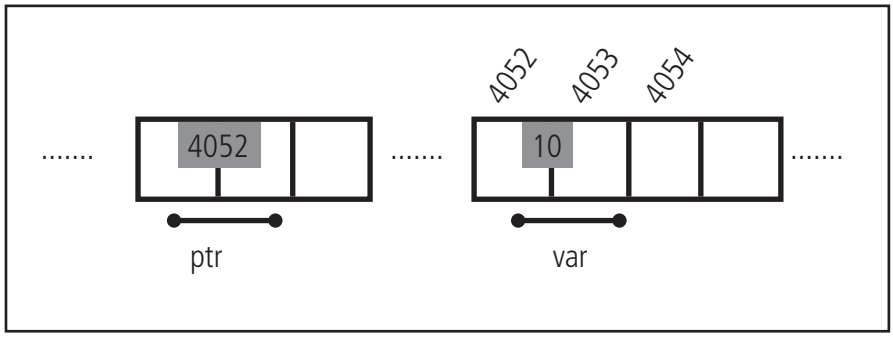


Figura 2.1: Inicialização de ponteiro

```
int var;

int *ptr;

var = 10;

ptr = &var;
```

Na sequência acima, são declarados uma variável tipo *int (var)* e um ponteiro para o mesmo tipo (*ptr*). A terceira linha atribui o valor 10 a var e a última linha inicializa o ponteiro *ptr*.

Observa-se o uso do operador de endereçamento (&) para inicialização do ponteiro. Isso significa, no código ptr = &var, que **ptr** passa a conter o endereço de var, não o seu valor. Supondo que o sistema é endereçado por 2 *bytes*, a Figura 2.1 acima dá uma ideia gráfica dessa associação: o conteúdo de *ptr* é 4052, que é o endereço do primeiro *byte* da variável var (*ptr* ocupa 2 *bytes* por causa do endereçamento do sistema e *var* também ocupa 2 *bytes*, mas por ser do tipo ***int***.

O valor 4052 para a posição de memória de var é apenas ilustrativo. Na prática, dependerá do local de memória onde o programa foi carregado.

Com *ptr* apontando para *var*, é possível realizar operações com esta última de forma indireta, a partir de *ptr*. Exemplos a seguir.

Acrescentando a linha

```
int newVar = *ptr;
```

ao código anterior, o valor de *newVar* é 10, que é o valor de var lido indiretamente através de *ptr*.

E a linha

```
*ptr = 20;
```

modifica o valor de *var* para 20, ou seja, altera de forma indireta através de *ptr*.

É importante lembrar que um ponteiro declarado e não inicializado poderá ter conteúdo nulo ou aleatório, a depender da alocação sistema. Nessa condição, se o conteúdo apontado for modificado, algumas posições de memória terão seus valores indevidamente alterados e as consequências serão imprevisíveis.

Agora veremos um exemplo completo de utilização de ponteiros:

```
#include <stdio.h>
#include <stdlib.h>
int main ( ) {
 int a;
 int b;
 int c;
 int *ptr; /* declara um ponteiro para um inteiro */
 /* um ponteiro para uma variável do tipo inteiro */
 a = 100;
 b = 2;
 c = 3;
 ptr = &a; /* ptr recebe o endereço da variável a */
 printf("Valor de ptr: %p, Conteúdo de ptr: %d\n", ptr, *ptr);
 printf("B: %d, C: %d \n", b, c);
 a = 200;
 printf("Valor de ptr: %p, Conteúdo de ptr: %d\n", ptr, *ptr);
 system("pause");
}
```

Escreva um resumo sobre ponteiros. Poste no AVEA da disciplina o arquivo de sua atividade.

2.4 Arquivos

Um arquivo pode ser visto de duas maneiras, na maioria dos sistemas operacionais: em "modo texto", como um texto composto de uma sequência de caracteres, ou em "modo binário", como uma sequência de *bytes* (números binários). Podemos optar por salvar (e recuperar) informações em disco usando um dos dois modos, texto ou binário. Uma vantagem do arquivo texto é que pode ser lido por uma pessoa e editado com editores de textos convencionais. Em contrapartida, com o uso de um arquivo binário é possível salvar (e recuperar) grandes quantidades de informação de forma bastante eficiente. O sistema operacional pode tratar arquivos "texto" de maneira diferente da utilizada para tratar arquivos "binários". Em casos especiais, pode ser interessante tratar arquivos de um tipo como se fossem do outro, tomando os cuidados apropriados.

2.4.1 Funções para manipulação de arquivos

A função básica para abrir um arquivo é *fopen*:

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

A-Z

FILE
É um tipo definido pela biblioteca padrão que representa uma abstração do arquivo.

Quando abrimos um arquivo, a função tem como valor de retorno um ponteiro para o tipo ***FILE***, e todas as operações subsequentes nesse arquivo receberão este endereço como parâmetro de entrada. Se o arquivo não puder ser aberto, a função tem como retorno o valor *NULL*.

Devemos passar o nome do arquivo a ser aberto. O nome do arquivo pode ser relativo, e o sistema procura o arquivo a partir do diretório corrente (diretório de trabalho do programa), ou pode ser absoluto, onde especificamos o nome completo do arquivo, incluindo os diretórios, desde o diretório raiz. Existem diferentes modos de abertura de um arquivo. Podemos abrir um arquivo para leitura ou para escrita, e devemos especificar se o arquivo será aberto em modo texto ou em modo binário. O parâmetro modo da função *fopen* é uma cadeia de caracteres onde espera-se a ocorrência de caracteres que identificam o modo de abertura. Os caracteres interpretados no modo são:

R	read-only Indica modo apenas para leitura, não pode ser alterado.
W	write Indica modo para escrita.
A	append Indica modo para escrita ao final do existente.
T	text Indica modo texto.
B	binary Indica modo binário.

Se o arquivo já existe e solicitamos a sua abertura para escrita com modo *w*, o arquivo é apagado e um novo, inicialmente vazio, é criado. Quando solicitamos com modo a, o mesmo é preservado e novos conteúdos podem ser escritos no seu fim. Com ambos os modos, se o arquivo não existe, um novo é criado.

Os modos *b* e *t* podem ser combinados com os demais. Maiores detalhes e outros modos de abertura de arquivos podem ser encontrados nos manuais da linguagem C. Em geral, quando abrimos um arquivo, testamos o sucesso da abertura antes de qualquer outra operação, por exemplo:

```
FILE* fp;
fp = fopen("entrada.txt","rt");
if (fp == NULL) {
printf("Erro na abertura do arquivo!\n");
exit(1);
}
```

Após ler/escrever as informações de um arquivo, devemos fechá-lo. Para fechar um arquivo, devemos usar a função *fclose*, que espera como parâmetro o ponteiro do arquivo que se deseja fechar. O protótipo da função é:

```
int fclose (FILE* fp);
```

O valor de retorno dessa função é zero, se o arquivo for fechado com sucesso, ou a constante *EOF* (definida pela biblioteca), que indica a ocorrência de um erro.

2.4.2 Arquivos em modo texto

Nesta seção, vamos descrever as principais funções para manipular arquivos em modo texto. Também discutiremos algumas estratégias para organização de dados em arquivos.

2.4.3 Funções para ler dados

A principal função de C para leitura de dados em arquivos em modo texto é a função *fscanf*, similar à função *scanf* que temos usado para capturar valores entrados via o teclado. No caso da *fscanf*, os dados são capturados de um arquivo previamente aberto para leitura. A cada leitura, os dados correspondentes são transferidos para a memória e o ponteiro do arquivo

avança, passando a apontar para o próximo dado do arquivo (que pode ser capturado numa leitura subsequente). O protótipo da função *fscanf* é:

```
int fscanf (FILE* fp, char* formato, ...);
```

Conforme pode ser observado, o primeiro parâmetro deve ser o ponteiro do arquivo do qual os dados serão lidos. Os demais parâmetros são os já discutidos para a função *scanf*: o formato e a lista de endereços de variáveis que armazenarão os valores lidos. Como a função *scanf*, a função *scanf* também tem como valor de retorno o número de dados lidos com sucesso.

Uma outra função de leitura muito usada em modo texto é a função *fgetc* que, dado o ponteiro do arquivo, captura o próximo caractere do arquivo. O protótipo dessa função é:

```
int fgetc (FILE* fp);
```

Apesar do tipo do valor de retorno ser int, o valor retornado é o caractere lido. Se o fim do arquivo for alcançado, a constante *EOF* (end of file) é retornada. Outra função muito utilizada para ler linhas de um arquivo é a função *fgets*. Essa função recebe como parâmetros três valores: a cadeia de caracteres que armazenará o conteúdo lido do arquivo, o número máximo de caracteres que deve ser lido e o ponteiro do arquivo. O protótipo da função é:

```
char* fgets (char* s, int n, FILE* fp);
```

A função lê do arquivo uma sequência de caracteres, até que um caractere **'\n'** seja encontrado ou que o máximo de caracteres especificado seja alcançado. A especificação de um número máximo de caracteres é importante para evitarmos que se invada memória quando a linha do arquivo for maior do que supúnhamos. Assim, se dimensionarmos nossa cadeia de caracteres, que receberá o conteúdo da linha lida, com 121 caracteres, passaremos esse valor para a função, que lerá no máximo 120 caracteres, pois o último será ocupado pelo finalizador de string – o caractere **'\0'**. O valor de retorno dessa função é o ponteiro da própria cadeia de caracteres passada como parâmetro ou *NULL* no caso de ocorrer erro de leitura (por exemplo, quando alcançar o final do arquivo).

2.4.4 Funções para escrever dados

Dentre as funções que existem para escrever (salvar) dados em um arquivo, vamos considerar as duas mais frequentemente utilizadas: *fprintf* e *fputc*, que são análogas, mas para escrita, às funções que vimos para leitura.

A função *fprintf* é análoga a função *printf* que temos usado para imprimir dados na saída padrão – em geral, o monitor. A diferença consiste na presença do parâmetro que indica o arquivo para o qual o dado será salvo. O valor de retorno dessa função representa o número de *bytes* escritos no arquivo. O protótipo da função é dado por:

```
int fprintf(FILE* fp, char* formato, ...);
```

A função *fputc* escreve um caractere no arquivo. O protótipo é:

```
int fputc (int c, FILE* fp);
```

O valor de retorno dessa função é o próprio caractere escrito, ou *EOF* se ocorrer um erro na escrita.

2.4.5 Estruturação de dados em arquivos textos

Existem diferentes formas para estruturarmos os dados em arquivos em modo texto, e diferentes formas de capturarmos as informações contidas neles. A forma de estruturar e a forma de tratar as informações dependem da aplicação. A seguir, apresentaremos três formas de representarmos e acessarmos dados armazenados em arquivos: caractere a caractere, linha a linha, e usando palavras chaves.

2.4.6 Acesso caractere a caractere

Para exemplificar o acesso caractere a caractere, vamos discutir duas aplicações simples. Inicialmente, vamos considerar o desenvolvimento de um programa que conta as linhas de um determinado arquivo (para simplificar, vamos supor um arquivo fixo, com o nome ***"entrada.txt"***. Para calcular o número de linhas do arquivo, podemos ler, caractere a caractere, todo o conteúdo do arquivo, contando o número de ocorrências do caractere que indica mudança de linha, isto é, o número de ocorrências do caractere '\n'.

```
/* Conta número de linhas de um arquivo */
#include <stdio.h>
int main (void) {
int c;
int nlinhas = 0; /* contador do número de linhas */
FILE *fp;
/* abre arquivo para leitura */
fp = fopen("entrada.txt","rt");
if (fp==NULL) {
printf("Não foi possivel abrir arquivo.\n");
return 1;
}
/* lê caractere a caractere */
while ((c = fgetc(fp)) != EOF) {
if (c == '\n')
nlinhas++;
}
/* fecha arquivo */
fclose(fp);
/* exibe resultado na tela */
printf("Numero de linhas = %d\n", nlinhas);
return 0;
}
```

Como segundo exemplo, vamos considerar o desenvolvimento de um programa que lê o conteúdo do arquivo e cria um arquivo com o mesmo conteúdo, mas com todas as letras minúsculas convertidas para maiúsculas. Os nomes dos arquivos serão fornecidos, via teclado, pelo usuário. Uma possível implementação desse programa é mostrada a seguir:

```
/* Converte arquivo para maiúsculas */
#include <stdio.h>
#include <ctype.h> /* função toupper */
int main (void) {
int c;
char entrada[121]; /* armazena nome do arquivo de entrada */
char saida[121]; /* armazena nome do arquivo de saída */
FILE* e; /* ponteiro do arquivo de entrada */
FILE* s; /* ponteiro do arquivo de saída */
/* pede ao usuário os nomes dos arquivos */
printf("Digite o nome do arquivo de entrada: ");
scanf("%120s",entrada);
printf("Digite o nome do arquivo de saida: ");
scanf("%120s",saida);
/* abre arquivos para leitura e para escrita */
e = fopen(entrada,"rt");
if (e == NULL) {
printf("Não foi possível abrir arquivo de entrada.\n");
return 1;
}
s = fopen(saida,"wt");
if (s == NULL) {
printf("Não foi possível abrir arquivo de saida.\n");
fclose(e);
return 1;
}
```

```
/* lê da entrada e escreve na saída */

while ((c = fgetc(e)) != EOF)

fputc(toupper(c),s);

/* fecha arquivos */

fclose(e);

fclose(s);

return 0;

}
```

2.4.7 Acesso linha a linha

Em diversas aplicações, é mais adequado tratar o conteúdo do arquivo linha a linha. Um caso simples que podemos mostrar consiste em procurar a ocorrência de uma sub-cadeia de caracteres dentro de um arquivo (análogo a o que é feito pelo utilitário *grep* dos sistemas *Unix*). Se a sub-cadeia for encontrada, apresentamos como saída o número da linha da primeira ocorrência. Para implementar esse programa, vamos utilizar a função *strstr*, que procura a ocorrência de uma sub-cadeia numa cadeia de caracteres maior. A função retorna o endereço da primeira ocorrência ou *NULL*, se a sub-cadeia não for encontrada. O protótipo dessa função é:

```
char* strstr (char* s, char* sub);
```

A nossa implementação consistirá em ler, linha a linha, o conteúdo do arquivo, contanto o número da linha. Para cada linha, verificamos se a ocorrência da sub-cadeia, interrompendo a leitura em caso afirmativo.

```
/* Procura ocorrência de sub-cadeia no arquivo */

#include <stdio.h>

#include <string.h> /* função strstr */

int main (void) {

int n = 0; /* número da linha corrente */

int achou = 0; /* indica se achou sub-cadeia */

char entrada[121]; /* armazena nome do arquivo de entrada */

char subcadeia[121]; /* armazena sub-cadeia */

char linha[121]; /* armazena cada linha do arquivo */
```

```
FILE* fp; /* ponteiro do arquivo de entrada */
/* pede ao usuário o nome do arquivo e a sub-cadeia */
printf("Digite o nome do arquivo de entrada: ");
scanf("%120s",entrada);
printf("Digite a sub-cadeia: ");
scanf("%120s",subcadeia);
/* abre arquivos para leitura */
fp = fopen(entrada,"rt");
if (fp == NULL) {
printf("Não foi possível abrir arquivo de entrada.\n");
return 1;
}
/* lê linha a linha */
while (fgets(linha,121,fp) != NULL) {
n++;
if (strstr(linha,subcadeia) != NULL) {
achou = 1;
break;
}
}
/* fecha arquivo */
fclose(fp);
/* exibe saída */
if (achou)
printf("Achou na linha %d.\n", n);
else
printf("Nao achou.");
return 0;
```

2.4.8 Acesso via palavras chave

Quando os objetos num arquivo têm descrições de tamanhos variados, é comum adotarmos uma formatação com o uso de palavras chave. Cada objeto é precedido por uma palavra chave que o identifica. A interpretação desse tipo de arquivo pode ser feita lendo-se as palavras chave e interpretando a descrição do objeto correspondente. Para ilustrar, vamos considerar que, além de retângulos, triângulos e círculos, também temos polígonos quaisquer no nosso conjunto de figuras geométricas. Cada polígono pode ser descrito pelo número de vértices que o compõe, seguido das respectivas coordenadas desses vértices.

O fragmento de código a seguir ilustra a interpretação desse formato, onde *fp* representa o ponteiro para o arquivo aberto para leitura.

```
...
FILE* fp;
char palavra[121];
...
while (fscanf(fp,"%120s",palavra) == 1) {
        if (strcmp(palavra,"RETANGULO")==0) {
                /* interpreta retângulo */
        }
        else
                if (strcmp(palavra,"TRIANGULO")==0) {
                        /* interpreta triângulo */
                }
                else
                        if (strcmp(palavra,"CIRCULO")==0) {
                                /* interpreta círculo */
                        }
                        else
```

```
                    if (strcmp(palavra,"POLIGONO")==0) {
                        /* interpreta polígono */
                    }
                    else {   /* trata erro de formato */
                    }
}
```

2.4.9 Arquivos em modo binário

Arquivos em modo binário servem para salvarmos (e depois recuperarmos) o conteúdo da memória principal diretamente no disco. A memória é escrita copiando-se o conteúdo de cada *byte* da memória para o arquivo. Uma das grandes vantagens de se usar arquivos binários é que podemos salvar (e recuperar) uma grande quantidade de dados de forma bastante eficiente. Neste curso, vamos apenas apresentar as duas funções básicas para manipulação de arquivos binários.

2.4.10 Função para salvar e recuperar

Para escrever (salvar) dados em arquivos binários, usamos a função *fwrite*. O protótipo dessa função pode ser simplificado por:

```
int fwrite (void* p, int tam, int nelem, FILE* fp);
```

O primeiro parâmetro dessa função representa o endereço de memória cujo conteúdo deseja-se salvar em arquivo. O parâmetro *tam* indica o tamanho, em *bytes*, de cada elemento e o parâmetro *nelem* indica o número de elementos. Por fim, passa-se o ponteiro do arquivo binário para o qual o conteúdo da memória será copiado.

A função para ler (recuperar) dados de arquivos binários é análoga, sendo que agora o conteúdo do disco é copiado para o endereço de memória passado como parâmetro. O protótipo da função pode ser dado por:

```
int fread (void* p, int tam, int nelem, FILE* fp);
```

Para exemplificar a utilização dessas funções, vamos considerar que uma aplicação tem um conjunto de pontos armazenados num vetor. O tipo que define o ponto pode ser:

```
struct ponto {
float x, y, z;
};
typedef struct ponto Ponto;
```

Uma função para salvar o conteúdo de um vetor de pontos pode receber como parâmetros o nome do arquivo, o número de pontos no vetor, e o ponteiro para o vetor. Uma possível implementação dessa função é ilustrada abaixo:

```
void salva (char* arquivo, int n, Ponto* vet) {
FILE* fp = fopen(arquivo,"wb");
if (fp==NULL) {
printf("Erro na abertura do arquivo.\n");
exit(1);
}
fwrite(vet,sizeof(Ponto),n,fp);
fclose(fp);
}
```

Assista à video-aula da disciplina de Estrutura de Dados disponível em http://cefetpi.nucleoead.net/etapi/. Aproveite para revisar os conteúdos da aula sobre Funções, Matrizes, Ponteiros e Arquivos.

A rigor, os tipos *int* são substituídos pelo tipo *size_t*, definido pela biblioteca padrão, sendo, em geral, sinônimo para um inteiro sem sinal (*unsigned int*).

Resumo

Nesta aula abordamos as principais implementações da linguagem de programação C, aprendemos ainda como criar funções ou sub-rotinas, a manipulação de matrizes, a implementação de programas utilizando ponteiros. Por fim, aprendemos a criar novos tipo de dados implementado registro e manipulando arquivos. Além de exemplos práticos do nosso dia-a-dia, a fim de disponibilizar ao leitor um excelente embasamento teórico.

Atividades de aprendizagem

1. Fazer um programa que calcule o volume de uma esfera, sendo que o volume de uma esfera é raio*raio*raio. Crie uma função que faça esse cálculo.

2. Elabore programa que leia "n" números digitados e apresente sua média.

3. Com base no estudo de Ponteiros, qual das opções a baixo é correto afirmar?

   a) Ponteiro é uma variável que armazena endereço.

   b) Ponteiro é o valor de uma variável.

   c) Ponteiro é o indicador da próxima variável a ser passada.

   d) Ponteiro é o endereço que aponta para uma variável.

4. Quais das seguintes instruções declaram um ponteiro para uma variável *float*?

   a) *Float* p

   b) *Float* *p

   c) **Float* p

   d) *Float** p

5. O seguinte programa está correto? Justifique.

```
#include <stdio.h>

const int VAL=123;

int main() {

    int *p = VAL;

    printf("%d \n", *p);

    return 0;

}
```

Aula 3 – Visão geral de estrutura de dados e lista linares

Objetivos

Conceituar estrutura de dados.

Descrever os tipos de estrutura de dados.

Implementar as operações básicas da estrutura de dados do tipo lista estática e dinâmica de ordenação e desordenação.

3.1 Introdução

Em um projeto de *software*, existem dois aspectos que devem ser estudados: os procedimentos que devem ser previstos pelo *software* e sobre que dados estes procedimentos irão atuar.

Nas técnicas estruturadas de projeto de *software* era dada ênfase aos procedimentos, com a identificação dos aspectos funcionais na primeira etapa de desenvolvimento do *software*.

Com as técnicas para especificação dos dados a nível conceitual, a importância dos procedimentos e dados tornou-se equilibrada. Atualmente, já existem técnicas de programação com ênfase nos dados (programação baseada em objetos).

Mas, independentemente das técnicas de análise e programação utilizadas, programas precisam manipular dados e é extremamente importante o conhecimento de conceitos e detalhes da implementação das diversas estruturas de dados que manipulam estes dados.

Neste texto, inicialmente, alguns conceitos básicos são apresentados. A seguir, são descritas as principais Estruturas de Dados, com seus respectivos algoritmos.

3.2 Conceitos básicos

Nesta seção serão apresentados conceitos essenciais para o desenvolvimento desta disciplina: tipos de dados, tipos abstratos de dados e estruturas de dados.

3.2.1 Tipos de dados

Em computação precisamos identificar os tipos de dados que o computador, a linguagem de programação ou mesmo um algoritmo são capazes de entender. De uma forma geral, os tipos de dados são diferenciados pelos valores que podem assumir e pelo conjunto de operações que podemos efetuar com eles.

Em linguagens de programação o tipo de dados de uma variável define o conjunto de valores que esta variável pode assumir. Uma variável do tipo lógico, por exemplo, pode assumir dois valores: verdadeiro ou falso. As possibilidades do *hardware* são previstas pela linguagem.

Os tipos de dados são divididos em: tipos primitivos de dados e tipos estruturados de dados.

Os tipos primitivos de dados são os tipos básicos, a partir dos quais podemos definir os demais tipos e estruturas de dados. Estes tipos de dados são os mais frequentes nas linguagens de programação e tem um conjunto de valores e operações restrito.

São considerados tipos primitivos de dados:

I. Inteiro - representa uma quantidade "que pode ser contada".

II. Real - representa um valor que pode ser fracionado.

III. Lógico - pode representar dois estados (verdadeiro ou falso).

IV. Caracter - pode representar dígitos, letras ou sinais.

V. Ponteiro - representa o endereço de um dado na memória.

Os tipos estruturados de dados são construídos a partir dos tipos primitivos. Estes tipos são previstos por muitas linguagens de programação e devem ser definidos pelo programador. Exemplos de tipos estruturados: *array* e registro. Estes dois tipos são formados por tipos básicos como inteiros, caracteres, reais, etc.

Uma declaração de variável em uma linguagem de programação como C especifica duas coisas:

I. Quantos *bytes* devem ser reservados para armazenar esta variável (Ex.: no caso de uma variável inteira, deve ser reservado um espaço que garanta que o maior inteiro permitido poderá ser representado).

II. Como estes *bytes* devem ser interpretados (Ex.: uma cadeia de *bits* pode ser interpretada como um inteiro ou um real).

Assim, tipos de dados podem ser vistos como métodos para interpretar o conteúdo da memória do computador.

3.2.2 Tipos abstratos de dados

O conceito de tipo de dados pode ser visto de outra perspectiva: levando em conta, não o que um computador pode fazer, mas o que os usuários desejam fazer. Este conceito de tipo de dados independente do *hardware* é chamado de Tipo Abstrato de Dados - TAD.

Um tipo abstrato de dados é composto por um modelo matemático acompanhado de um conjunto de operações definidas sobre este modelo.

Uma vez que um TAD é definido e as operações associadas a este tipo são especificadas, pode-se implementar este tipo de dado.

3.2.3 Estruturas de dados

Um algoritmo é projetado em termos de tipos abstratos de dados. Para implementá-los numa linguagem de programação é necessário encontrar uma forma de representá-los nessa linguagem, utilizando tipos e operações suportadas pelo computador. Para representar o modelo matemático do TAD, em uma linguagem de programação, emprega-se uma estrutura de dados.

As estruturas de dados diferem umas das outras pela disposição ou manipulação de seus dados. A disposição dos dados em uma estrutura obedece a condições preestabelecidas e caracteriza a estrutura.

Assim, Estrutura de Dados é um método particular de se implementar um TAD. A implementação de um TAD escolhe uma estrutura de dados (ED) para representá-lo. Cada ED pode ser construída a partir de tipos básicos (inteiro, real, caracter) ou estruturada (*array*, registro) de uma determinada linguagem de programação.

O estudo de estruturas de dados não pode ser desvinculado de seus aspectos algorítmicos. A escolha correta da estrutura adequada a cada caso depende

diretamente do conhecimento de algoritmos para manipular a estrutura de maneira eficiente.

As estruturas de dados de tipos de dados estruturadas se dividem em homogêneos (vetores e matrizes) e heterogêneos (registros).

As estruturas homogêneas são conjuntos de dados formados pelo mesmo tipo de dado primitivo. E as estruturas heterogêneas são conjuntos de dados formados por tipos de dados primitivos diferentes (campos do registro) em uma mesma estrutura. A escolha de uma estrutura de dados apropriada pode tornar um problema complicado em um de solução bastante trivial. O estudo das estruturas de dados está em constante desenvolvimento (assim como o de algoritmos), mas, apesar disso, existem certas estruturas clássicas que se comportam como padrões.

Estruturas de dados clássicas:

I. Lista.

II. Pilha.

III. Fila.

IV. Árvores.

Nas próximas seções serão apresentados as principais Estruturas de Dados utilizadas para representar listas, pilhas, filas e árvores. Além disso, serão apresentados os algoritmos que devem ser utilizados para manipular estas estruturas.

3.2.4 Listas lineares

Uma das formas mais comumente usadas para se manter dados agrupados é a lista. Afinal, quem nunca organizou uma lista de compras antes de ir ao mercado, ou então uma lista dos amigos que participarão da festa? As listas têm-se mostrado um recurso bastante útil e eficiente no dia-a-dia das pessoas. Em computação, não tem sido diferente: a lista é uma das estruturas de dados mais empregadas no desenvolvimento de programas.

Ao desenvolver uma implementação para listas lineares, o primeiro problema que surge é: como podemos armazenar os elementos da lista, dentro do computador?

Sabemos que antes de executar um programa, o computador precisa carregar seu código executável para a memória. Da área de memória que é reservada para o programa, uma parte é usada para armazenar as instruções a serem executadas e a outra é destinada ao armazenamento dos dados. Quem determina quanto de memória será usado para as instruções é o compilador. Alocar área para armazenamento de dados, entretanto, é responsabilidade do programador.

Uma lista linear pode ser implementada usando vetores ou ponteiros. Se for implementada usando vetores, deve estipular qual a quantidade de elementos que a lista pode armazenar. A memória para armazenamento dos dados é alocada em tempo de compilação. Quando implementada usando ponteiros, a memória é alocada conforme novos elementos são colocados na lista e desalocada quando elementos são retirados. Ou seja, vai alocando memória dinamicamente, em tempo de execução.

3.2.4.1 Tipos de listas lineares

I. Lista estática desordenada.

II. Lista estática ordenada.

III. Lista dinâmica desordenada.

IV. Lista dinâmica ordenada.

a) Lista estática desordenada

Na lista desordenada os elementos são colocados na primeira posição vazia da lista (normalmente, no final). Na lista ordenada, é escolhido um dado que será o campo de ordenação da lista. Quando se deseja inserir um novo elemento na lista, primeiro tem que ser verificado em que local ele dever ser colocado para que seja mantida a ordem da lista.

Operações básicas das listas: inserir elemento, remover elemento, consultar elemento, alterar elemento, listagem dos elementos da lista.

Esta estrutura é implementada usando vetores e não se preocupa com ordenação. Os elementos são colocados na estrutura por ordem de chegada. Nas próximas seções será descrita cada uma das operações feitas nesta estrutura. No exemplo, teremos uma lista com os dados dos alunos de uma turma (para simplificar, cada aluno tem apenas a matrícula, nome e seu polo).

- **Operações básicas**

I. Inserir elemento

A Figura 3.1 ilustra a inserção de três elementos em uma lista estática desordenada. Inicialmente o vetor está vazio. O primeiro elemento a chegar é o aluno José com matrícula 256. Este será colocado na primeira posição do vetor.

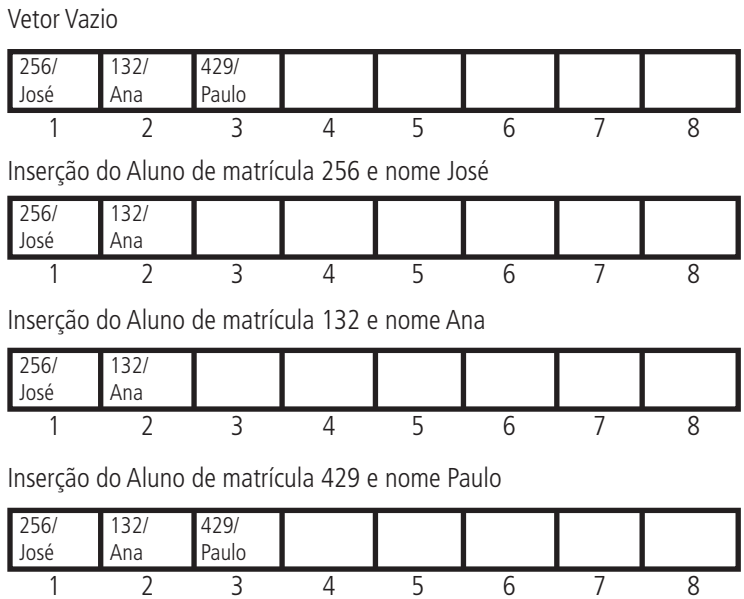


Figura 3.1: Inserção da lista estática desordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Posteriormente, chegam mais dois alunos (Ana com matrícula 132 e Paulo com matrícula 429), que são colocados nas próximas posições disponíveis do vetor (posição 2 e posição 3). Quando uma inserção vai ser executada, é necessário verificar se o vetor tem posições disponíveis. Caso contrário, a inserção não pode ser efetuada.

O código abaixo, implementado está em linguagem C, inserir um novo aluno à lista:

```
//Inserir novo aluno
void inserir() {
    int cont;
    do{
        cabec();
```

```
            printf("\nInserir Novo Aluno\n\n");
            if (qa < maximo) { // verifica se o vetor pode receber novo aluno
                printf("\nMatricula do Aluno: ");
                scanf("%d",&turma[qa].mat);
                printf("\nNome: ");
                fflush(stdin);
                gets(turma[qa].nome);
                printf("\nPolo: ");
                scanf("%s",&turma[qa].polo);
                qa++;
                printf("\n\nAluno Inserido com Sucesso!!!\n\n");
            }
            else { // vetor cheio
                    printf("\n\n\aNao Pode Inserir - Turma Cheia!!!\n\n");
                    getche();
                    break;
            }
            printf("\n\nInserir outro(1-sim/2-nao)? ");
            scanf("%d",&cont);
        }while (cont == 1);
}
```

II. Consultar elemento

Depois que um elemento é inserido, a operação mais executada é a consulta. Para a consulta é necessário saber qual elemento deseja consultar. Neste caso faremos uma consulta por matrícula. Para isso, a matrícula do aluno a ser consultado deve ser lida. É feita uma pesquisa em todas as posições ocupadas do vetor, a procura do elemento.

Vetor

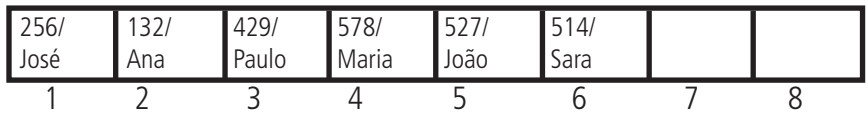


Figura 3.2: Consulta da lista estática desordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

No vetor da Figura 3.2 temos seis elementos. Vamos consultar o elemento de matrícula 578. Para encontrá-lo, temos que varrer o vetor desde o seu início e paramos na posição 4 que é a posição onde ele se encontra. Caso quiséssemos consultar o elemento de matrícula 192, iríamos varrer todo o vetor, elemento a elemento e ao chegar na sexta posição (que é a ultima) ficaríamos sabendo que este elemento não se encontra no vetor. Quando um elemento é encontrado, seus dados são apresentados e quando ele não está no vetor, uma mensagem de erro deve ser dada ao usuário.

O código abaixo, implementado está em linguagem C, realiza um consulta de um determinado aluno:

```
//Consultar aluno cadastrado por matricula
void consultarmat() {
    int i, matcon, achou, cont;
    do {
        cabec();
        printf("\nConsultar Aluno por Matricula\n\n");
        printf("\nMatricula do Aluno: ");
        scanf("%d",&matcon);
        achou = procura(matcon);
        if (achou != -1)
            mostre(achou);
        else // aluno nao foi encontrado
            printf("\n\n\aNumero de Matricula Incorreto!!!!!!\n");
        printf("\n\nConsultar outro(1-sim/2-nao)? ");
        scanf("%d",&cont);
    } while (cont == 1);
}
```

III. Remover elemento

Caso um elemento não precise mais fazer parte da estrutura, ele pode ser removido. Para remover os dados de um elemento é necessário saber qual elemento deseja remover. Já que iremos Neste caso faremos a busca por matrícula. Para isso, a matrícula do aluno a ser removido deve ser lida. É feita uma pesquisa em todas as posições ocupadas do vetor, a procura da matrícula. Assim que o elemento é encontrado, seus dados devem ser apresentados ao usuário (neste caso a matrícula e o nome).

Dessa forma ele pode verificar se realmente é aquele o elemento a ser removido. Quando o elemento não é encontrado, uma mensagem de erro deve ser dada ao usuário. Numa lista desordenada, para a remoção ser efetuada, o último elemento do vetor deve ser transferido para a posição do elemento removido, conforme Figura 3.3 a seguir.

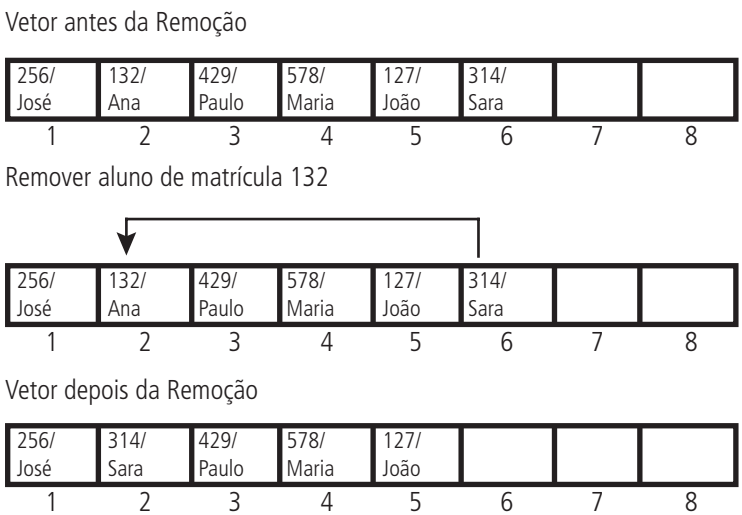


Figura 3.3: Remoção da lista estática desordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código abaixo, implementado está em linguagem C, realiza uma remoção aluno:

```
//Remover aluno cadastrado
void remover() {
    int matrem, i, cont, achou, conrem;
    do{
        cabec();
        printf("\nRemover Aluno\n\n");
```

```
            printf("\nMatricula do Aluno: ");
            scanf("%d",&matrem);
            achou = procura(matrem);
            if (achou != -1) {
                mostre(achou);
                printf("\nDeseja remover o aluno (1-sim/2-nao)? ");
                scanf("%d",&conrem);
                if (conrem==1) { // verifica se quer remover
                    turma[i]= turma[qa-1];
                    qa--;
                    printf("\n\nAluno removido com Sucesso!!!\n");
                }
                else
                    printf("\n\n\aO aluno nao foi removido!!!\n");
                break;
            }
            else // aluno nao foi encontrado
                printf("\n\naNumero de Matricula Incorreto!!!!!!\n");
            printf("\n\nRemover outro(1-sim/2-nao)? ");
            scanf("%d",&cont);
        }while (cont == 1);
    }
```

IV. Listagem de todos os elementos

A operação de listagem possibilita a visualização dos dados de todos os elementos cadastrados. É feita uma varredura no vetor e todos os dados de todos os elementos são apresentados ao usuário. Caso a lista esteja vazia, será apresentada uma mensagem.

O código abaixo representa a impressão de todos os elementos:

```
//Imprimir relatório com as informações de todos alunos
void listagem() {
    int i;
    cabec();
    printf("\nRelatorio Geral\n");
    printf("\n\nMatricula Aluno Polo ");
    printf("\n-----------------------------------------");
    for(i = 0; i < qa; i++)
        printf("\n%9d %-20s %-10s", turma[i].mat, turma[i].nome,
turma[i].polo);
    printf("\n-----------------------------------------");
    printf("\n\nDigite qualquer tecla para sair... ");
    getche();
}
```

A implementação do código completo de lista estática desordenada na linguagem C estará disponibilizado no AVEA em formatos **.pdf** e **.cpp**!

b) Lista estática ordenada

Nesse tipo de lista, os elementos devem ser colocados na estrutura obedecendo a uma ordenação. Qualquer operação feita na estrutura, não pode afetar na ordenação da mesma. A vantagem dessa lista será notada principalmente nos momentos que necessite percorrer a lista a procura de algum elemento.

Aqui, o programa anterior será modificado, fazendo com que os alunos sejam armazenados no vetor por ordem de matrícula.

Esta estrutura é implementada usando vetores e se preocupa com ordenação. Os elementos são colocados na estrutura usando uma função procurando a posição correta de inserção na ordem crescente. Nas próximas seções será descrita cada uma das operações feitas nesta estrutura. No exemplo, teremos uma lista com os dados dos alunos de uma turma (para simplificar, cada aluno tem apenas a matrícula, nome e seu polo).

- **Operações básicas**

I. Inserir elemento

Para inserir um elemento em uma lista ordenada podem ocorrer cinco possibilidades:

I. A lista está cheia: nesse caso a inserção é cancelada.

II. A lista está vazia: o elemento é colocado na primeira posição do vetor.

III. O elemento a ser inserido é menor do que o primeiro da lista.

IV. O elemento a ser inserido é maior do que o ultimo da lista.

V. O elemento novo será inserido entre elementos da lista.

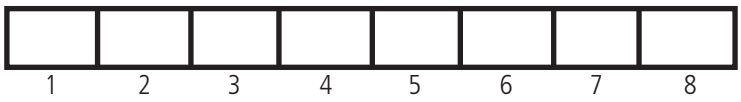
A Figura 3.4 ilustra a inserção de quatro elementos em uma lista estática ordenada. Inicialmente o vetor está vazio e o primeiro elemento a chegar é o aluno José com matrícula 256, conforme a Figura 3.4a e está inserção será colocada na primeira posição do vetor. Posteriormente, conforme Figura 34.b, chega Ana com matrícula 132.

Para que a lista fique ordenada, deve verificar em qual posição o elemento deve ser inserido. Nesse caso, a matrícula 132 é menor que 256 (que é a primeira da lista). Assim, todos os elementos a partir da posição onde o elemento de matrícula 256 se encontra, devem se deslocar uma posição, abrindo espaço para o elemento ser inserido.

Agora de acordo com a Figura 3.4c chega o elemento de matrícula 429. Esta matrícula é maior do que todas as matrículas do vetor, portanto, ele é inserido no final do vetor, sem necessidade de deslocamentos. E por final, chega o elemento de matrícula 197 que será inserido no espaço conformeFigura 3.4d.

Devemos descobrir onde se encontra o primeiro elemento maior do que o que será inserido, ou seja, seu sucessor imediato. Neste caso, ele deve entrar na posição do elemento de matrícula 256. O espaço deve ser liberado, fazendo-se necessário o deslocamento de todos os elementos a partir do elemento de matrícula 256. A principal questão da inserção ordenada é descobrir o local onde o elemento deve ser inserido e, se necessário, fazer os deslocamentos.

Vetor Vazio



Inserção do Aluno de matrícula 256 e nome José

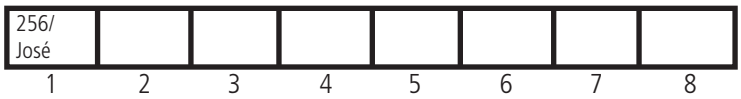


Figura 3.4a: Inserção da lista estática ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Inserção do Aluno de matrícula 132 e nome Ana

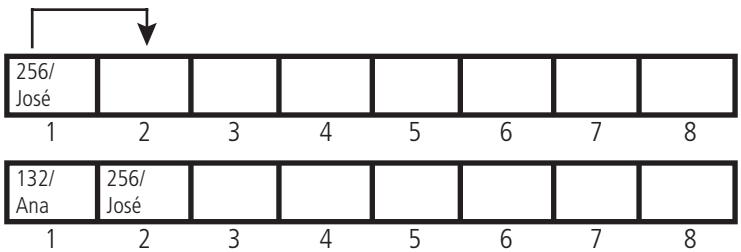


Figura 3.4b: Inserção da lista estática ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Inserção do Aluno de matrícula 429 e nome Paulo

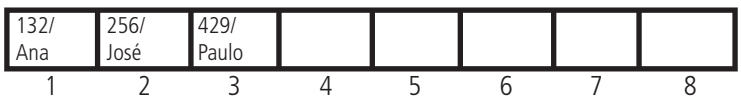


Figura 3.4c: Inserção da lista estática ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Inserção do Aluno de matrícula 197 e nome Maria

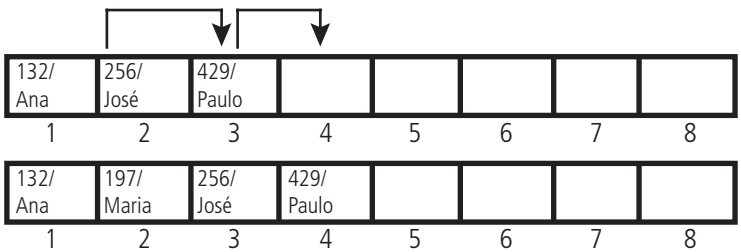


Figura 3.4d: Inserção da lista estática ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código a seguir, implementado está em linguagem C, inseri um novo aluno à lista:

```
// Inserir um novo aluno na escola
void inserir() {
    int cont;
    do{
        cabec();
        printf("\nInserir Novo Aluno\n");
        if (qa < maximo) { // verifica se o vetor pode receber novo aluno
            printf("\nMatricula do Aluno: ");
            scanf("%d",&al.mat);
            printf("\nNome: ");
            fflush(stdin);
            gets(al.nome);
            printf("\nPolo: ");
            scanf("%s",&al.polo);
            colocarordem();
            qa++;
            printf("\n\nAluno Inserido com Sucesso!!!\n");
        }
        else { // vetor cheio
            printf("\n\n\aNao Pode Inserir - Turma Cheia!!!\n");
            getche();
            break;
        }
        printf("\n\nContinuar inserindo aluno (1-sim/2-nao)? ");
        scanf("%d",&cont);
    }while (cont == 1);
}
```

II. Consultar elemento

Na Figura 3.5 a matrícula do aluno a ser consultado deve ser lida. É feita uma varredura em todas as posições ocupadas do vetor, a procura da matrícula. No caso de uma lista ordenada, se estivéssemos procurando um aluno de matrícula 120, assim que fizéssemos a leitura da primeira matrícula do vetor, já saberíamos que a matrícula 120 não está no vetor, evitando uma varredura até o final do vetor. Na estrutura ordenada, as consultas são mais eficientes.

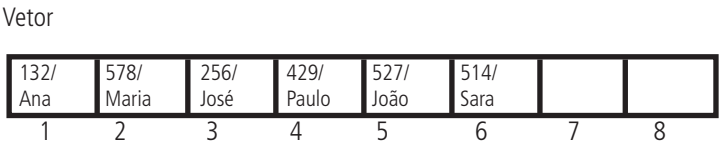


Figura 3.5: Consulta da lista estática ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código abaixo, implementado está em linguagem C, realiza um consulta de um determinado aluno:

```
//Consultar um aluno da escola
void consultar() {
    int matcon, achou, continuar;
    do{
        cabec();
        printf("\nConsultar Aluno\n\n");
        printf("\nMatricula do Aluno: ");
        scanf("%d",&matcon);
        achou = procura(matcon);
        if (achou!=-1)
            mostre(achou);
        else
            printf("\n\n\aNumero de Matricula Incorreto!!!!!!\n");
```

```
            printf("\n\nContinuar consultando aluno(1-sim/2-nao)? ");
            scanf("%d",&continuar);
        }while (continuar == 1);
}
```

III. Remover elemento

Começamos com a leitura da matrícula do aluno a ser removido. É feita uma varredura em todas as posições ocupadas do vetor, a procura da matrícula. Assim que o elemento é encontrado, seus dados devem ser apresentados ao usuário (neste caso a matrícula e o nome). Dessa forma ele pode verificar se realmente é aquele o elemento a ser removido. Quando o elemento não é encontrado, uma mensagem de erro deve ser dada ao usuário.

No exemplo da Figura 3.6 a seguir, desejamos remover o elemento de matrícula 256. Para que a lista fique contínua e ordenada, todos os elementos que vem depois do elemento que será removido, devem ser trazidos uma posição a frente.

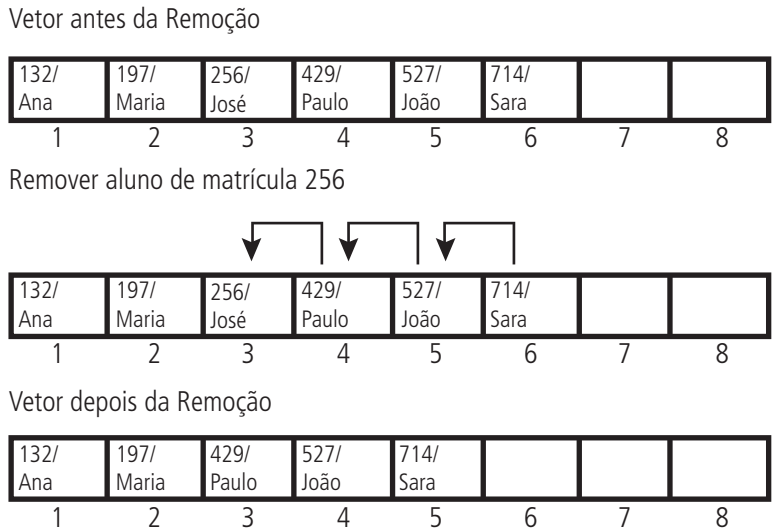


Figura 3.6: Remoção da lista estática ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código abaixo, implementado está em linguagem C, realiza uma remoção aluno:

```
// Remover um aluno da escola
void remover() {
    int matrem, i, achou, continuar, conrem;
    do{
        cabec();
        printf("\nRemover Aluno\n\n");
        printf("\nMatricula do Aluno: ");
        scanf("%d",&matrem);
        achou = procura(matrem);
        if (achou!=-1) {
            mostre(achou);
            printf("\nDeseja remover o aluno (1-sim/2-nao)? ");
            scanf("%d",&conrem);
            if (conrem == 1) {
                for (i=achou;i<qa;i++)
                    turma[i]=turma[i+1];
                qa--;
                printf("\n\nAluno removido com Sucesso!!!\n");
            }
            else
                printf("\n\n\aO aluno nao foi removido!!!\n");
        }
        else
            printf("\n\n\aNumero de Matricula Incorreto!!!!!!\n");
        printf("\n\nContinuar removendo aluno (1-sim/2-nao)? ");
```

```
            scanf("%d",&continuar);
        }while (continuar == 1);
}
```

IV. Listagem de todos os elementos

A operação de listagem possibilita a visualização dos dados de todos os elementos cadastrados. É feita uma varredura no vetor e todos os dados de todos os elementos são apresentados ao usuário.

O código abaixo representa a impressão de todos os elementos:

```
//Imprimir o relatorio contendo os dados de todos os alunos
void listagem() {
        int i;
        float soma=0;
        cabec();
        printf("\nRelatorio Geral\n");
        printf("\nMatricula Aluno Polo");
        printf("\n-----------------------------------------");
        for(i = 0; i<qa; i++)
 printf("\n%9d %-20s %-10s", turma[i].mat, turma[i].nome, turma[i].
polo);
        printf("\n-----------------------------------------");
        printf("\n\nDigite qualquer tecla para sair");
        getch();
}
```

A implementação do código completo de lista estática ordenada na linguagem C estará disponibilizado junto a plataforma AVEA em formatos **.pdf** e **.cpp**!

Vamos praticar, através do código completo do conteúdo visto até o momento da aula 3 disponível no AVEA, implemente um novo programa trocando variáveis alunos, matricula e polo por pessoa, rg e endereço.

c) Lista dinâmica desordenada

Esta lista é implementada usando ponteiros. A memória para armazenar os dados é alocada em tempo de execução. Na próxima seção será descrito o funcionamento de cada uma das operações básicas. O referido programa faz o cadastro dos alunos de uma turma em uma lista usando ponteiros. Esta lista possui dois ponteiros, um que guarda o endereço do primeiro elemento da lista ("inicio") e outro que guarda o endereço do último elemento da lista ("fim").

- **Operações básicas**

I. Inserir elemento

A Figura 3.7 ilustra a inserção de três elementos em uma lista dinâmica desordenada. Inicialmente a lista está vazia, portanto o valor do ponteiro inicio e fim é *NULL* (passo 1). Quando um elemento vai ser inserido, a memória é alocada e os dados são armazenados (passo 2). Todo nó criado em uma lista dinâmica desordenada, não tem vizinho seguinte, por isso ele aponta para *NULL*. Na inserção do primeiro elemento, os ponteiros **inicio** e **fim** apontam para este elemento (no exemplo, endereço 1010).

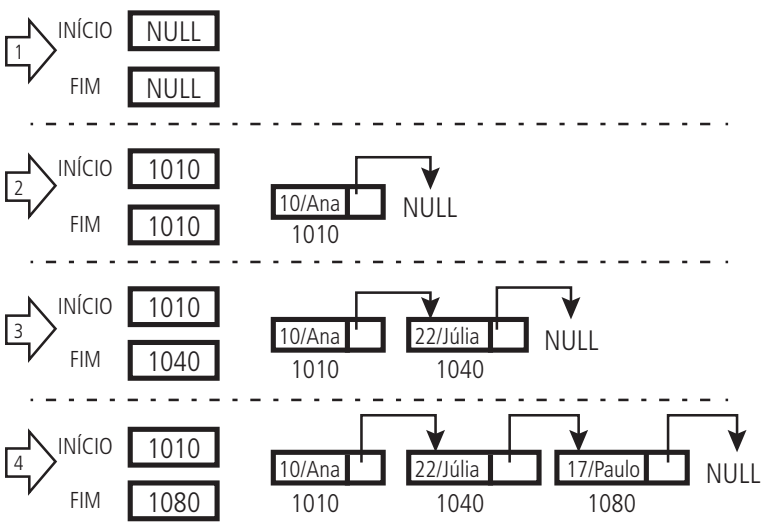


Figura 3.7: Inserção da lista dinâmica desordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

No (passo 3) temos a chegada de um outro elemento. A memória é alocada e o novo nó tem os dados preenchidos e tem que ser anexado aos outros nós da lista. Todo nó que chega aponta para *NULL* e o ponteiro **fim** passa a ter o endereço do nó que acabou de chegar. O nó que anteriormente era o último da lista (1010) passará a ter como vizinho o nó que acabou de chegar (ou seja, aponta para 1040).

O (passo 4) mostra uma outra inserção de nó. Veja que a cada novo elemento, apenas o endereço do ponteiro **fim** é modificado. O ponteiro **inicio** permanece com o mesmo valor.

O código abaixo, implementado está em linguagem C, insere um novo aluno à lista:

```
/* Funcao para inserir um novo no, ao final da lista */
void inserir () {
    TAluno *novono;
    int i, matl, continuar;
    char nomel[20];
    do{
        cabec();
        printf("\n Inserir novo aluno \n");
        printf("\n Matricula: ");
        scanf("%d",&matl);
        printf("\n Nome: ");
        fflush(stdin);
        gets(nomel);
        fflush(stdin);
        qa++;
```

```
            //Aloca memoria para o novo aluno e coloca os dados
            novono = (TAluno *)malloc(sizeof(TAluno));
            novono->mat = matl;
            for (i=0;i<=19;i++)
                  novono->nome[i] =nomel[i];
            novono->prox = NULL;
            // Inserir novono na lista de alunos
            if (inicio == NULL) {
                  inicio = novono;
                  fim = novono;
            }
            else {
                  fim->prox = novono;
                  fim = novono;
            }
            printf("\n\nInserido com Sucesso!!!!\n");
            printf("\nContinuar inserindo (1-sim/2-nao)? ");
            scanf("%d",&continuar);
      }while (continuar == 1);
}
```

II. Consultar elemento

Para fazer uma consulta em uma lista dinâmica é necessário saber qual elemento deseja consultar. Neste caso faremos uma consulta por matrícula. Um ponteiro **auxiliar** será usado para percorrer a lista, visitando cada nó a procura do elemento.

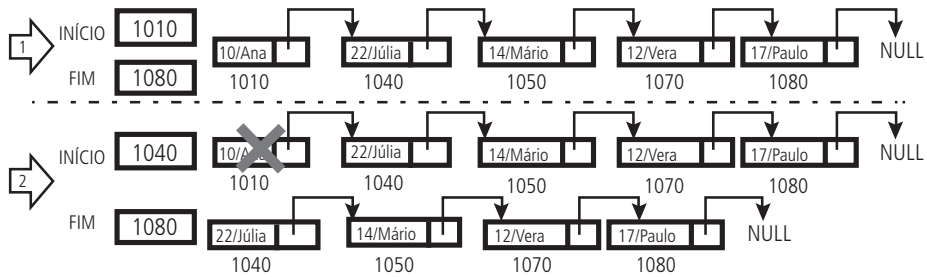


Figura 3.8: Consulta da lista dinâmica desordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Na Figura 3.8 temos uma lista com três elementos. Caso quiséssemos consultar o elemento de matrícula 25, iríamos percorrer a lista até chegar ao último nó, cujo endereço do vizinho é NULL (nó de endereço 1080) e ficaríamos sabendo que este elemento não se encontra na lista. Quando um elemento é encontrado, seus dados são apresentados e quando ele não está na lista, uma mensagem de erro deve ser dada ao usuário.

O código abaixo, implementado está em linguagem C, realiza um consulta de um determinado aluno:

```
/* Consultar um aluno na lista */
void consultar() {
    int matc, continuar, achou=0;
    do{
            cabec();
            printf("\nConsulta aluno pelo numero de matricula\n\n");
            printf("\nMatricula: ");
            scanf("%d",&matc);
            noatual = inicio;
            while(noatual != NULL) {
                    if (noatual->mat == matc) {
                    achou = 1;
                    printf("\n\nMatricula Nome\n");
```

```
                printf("---------------------------------------\n");
                printf("%9d %-20s\n",noatual->mat, noatual->nome);
                printf("---------------------------------------\n");
                break;
            }
            else
                noatual = noatual->prox;
        }
        if (achou == 0)
                printf("\n\nAluno nao encontrado!!\n");
          printf("\nContinuar consultando (1-sim/2-nao)? ");
          scanf("%d",&continuar);
    }while (continuar == 1);
}
```

III. Remover elemento

Para remover um elemento é necessário saber qual elemento deseja remover. Para isso, a matrícula do aluno a ser removido deve ser lida. É feita uma varredura em todos os nós da lista.

Assim que o elemento é encontrado, seus dados devem ser apresentados ao usuário (neste caso a matrícula e o nome). Dessa forma ele pode verificar se realmente é aquele o elemento a ser removido. Quando o elemento não é encontrado, uma mensagem de erro deve ser dada ao usuário.

Nas Figuras 3.9 são ilustrados os três casos de remoção: remover primeiro da lista, último da lista, elemento no meio da lista. No (passo 1), conforme Figura 3.9a, temos a lista com cinco elementos.

Já no (passo 2 também presente na Figura 3.9a foi feita a remoção do aluno com matrícula 10. Este é o primeiro nó da lista. Esta remoção é feita

modificando o valor do nó início para o endereço do vizinho do nó que está sendo removido, neste caso, endereço 1040.

Agora no (passo 3presente na Figura 3.9b foi removido a aluno de matrícula 17. Este aluno é o ultimo nó da lista, a sua remoção irá ter que atualizar o ponteiro fim. Além disso, o elemento que tinha como vizinho seguinte o nó de matrícula 17, terá agora *NULL* como vizinho. Portanto, dois ponteiros são atualizados.

Ainda na Figura 3.9b o (passo 4) realiza a última remoção, aluno com matrícula 14 (endereço 1050), que está armazenado em um nó no meio da lista. Nesta remoção, os ponteiros **inicio** e **fim** não são alterados. No entanto, o elemento que vem antes do removido (1040), terá agora como vizinho o elemento que era vizinho do que será removido (1070).

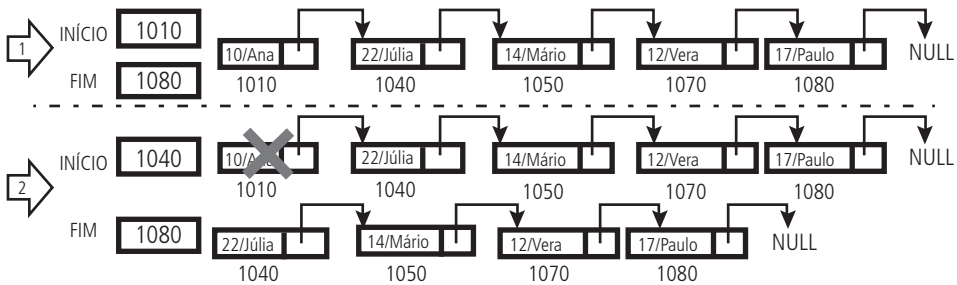


Figura 3.9a: Remoção da lista dinâmica desordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

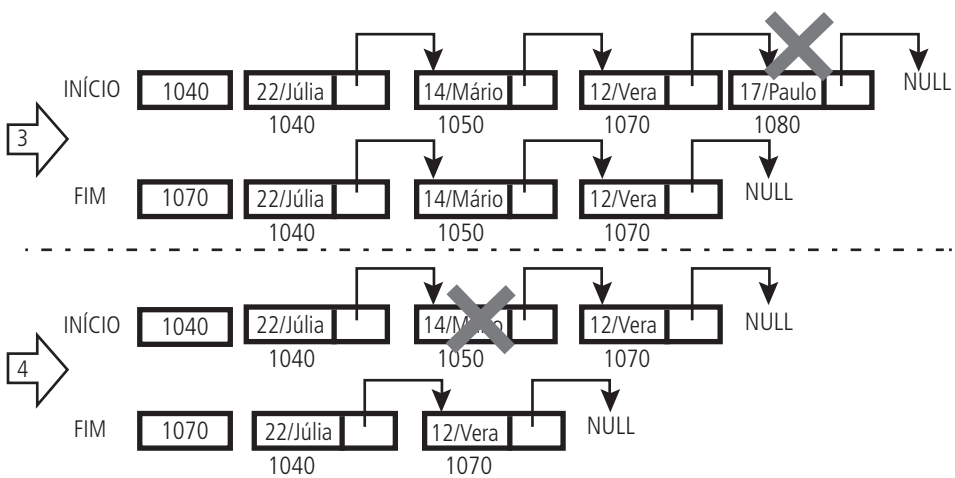


Figura 3.9b: Remoção da lista dinâmica desordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código a seguir, implementado está em linguagem C, realiza uma remoção aluno:

```
/* remover um aluno da lista */
void remover() {
 TAluno *noantrem;
 int matr, confrem, continuar, achou;
 do{
 achou = 0;
 cabec();
 printf("\nRemove aluno \n\n");
 printf("\nMatricula: ");
 scanf("%d",&matr);
 noatual = inicio;
 while(noatual != NULL) {
 if (noatual->mat == matr) {
 achou = 1;
 printf("\n\nMatricula Nome\n");
 printf("----------------------------------\n");
 printf("%9d %-20s\n",noatual->mat, noatual->nome);
 printf("----------------------------------\n");
 printf("\n\nDeseja remover o aluno (1-sim, 2-nao)? ");
 scanf("%d",&confrem);
 if (confrem ==1) {
 if (noatual == inicio)
 inicio = inicio->prox;
 else {
 noantrem->prox=noatual->prox;
 if (noatual == fim)
```

```
fim=noantrem;
}
qa--;
free(noatual);
printf("\n\nRemovido com sucesso!!!!\n");
}
else
printf("\n\nRemocao cancelada\n");
break;
}
else {
noantrem = noatual;
noatual = noatual->prox;
}
}
if (achou == 0)
printf("\n\nAluno nao encontrado!!\n");
printf("\n\nDeseja remover outro (1-sim, 2-nao)? ");
scanf("%d",&continuar);
}while (continuar ==1);
}
```

IV. Listagem de todos os elementos

A operação de listagem possibilita a visualização dos dados de todos os elementos cadastrados. É feita uma pesquisa na lista partindo do endereço **inicio** até o último nó, todos os dados de todos os elementos são apresentados ao usuário.

O código abaixo representa a impressão de todos os elementos:

```
/* Lista todos os alunos presentes na lista */
void listar () {
 noatual = inicio;
 cabec();
 printf("\nListagem de Alunos\n\n");
 if (qa != 0) {
 printf("\n\nMatricula Nome\n");
 printf("---------------------------------------\n");
 while( noatual != NULL) {
 printf("%9d %-20s\n",noatual->mat, noatual->nome);
 noatual = noatual->prox;
 }
 printf("---------------------------------------\n");
 printf("\n\nQuantidade de Alunos = %d\n",qa);
 }
 else
 printf("\n\n Nao tem nenhum aluno cadastrado");
 printf("\n\n\nTecle enter para voltar para o menu\n");
 getche();
}
```

A implementação do código completo da lista dinâmica desordenada na linguagem C estará disponibilizado junto ao AVEA em formatos . **pdf** e **.cpp**!

Vamos praticar, através do código completo do conteúdo visto até o momento da aula 3 disponível no AVEA, implemente um novo programa trocando variáveis alunos, matrícula e polo por pessoa, RG e endereço.

d) Lista dinâmica ordenada

Esta lista é implementada usando ponteiros, no entanto, quando for feito o caminhamento pelos nós da lista, ela deve estar ordenada por algum campo, neste exemplo o campo de ordenação é a matrícula. Esta lista possui apenas o ponteiro inicio, que guarda o endereço do primeiro elemento da lista.

- **Operações básicas**

I. Inserir elemento

Nas Figuras 3.10 (a e b) são ilustradas a inserção de quatro elementos em uma lista dinâmica ordenada. No (passo 1) presente na Figura 3.10a, a lista está vazia, com o ponteiro início apontando para *NULL*. Já no (passo 2) também presente na Figura 3.10a temos a inserção do elemento de matrícula 17. Como a lista está vazia, o ponteiro início vai apontar para este elemento (endereço 1080).

Continuando na Figura 3.10a, no (passo 3), temos a chegada de um outro elemento, matrícula 10. É verificado que ele tem a matrícula menor do que o primeiro elemento da lista, então este novo elemento terá que ser inserido no início da lista. Assim, o elemento novo vai apontar para o primeiro da lista e o ponteiro início irá apontar para o novo nó.

Agora observaremos a Figura 3.10b, o (passo 4) terá a inserção de um aluno com matrícula 14, que será inserido no meio da lista. Para isso, teremos que descobrir entre quais elementos ele irá ser inserido, para manter a lista ordenada e fazer as ligações dos ponteiros corretamente.

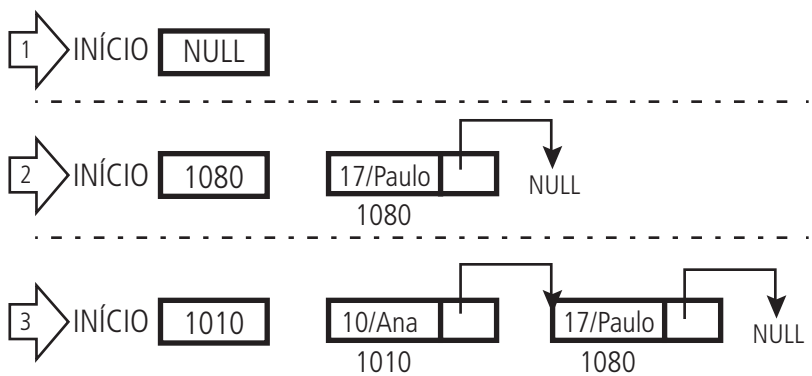


Figura 3.10a: Inserção da lista dinâmica ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

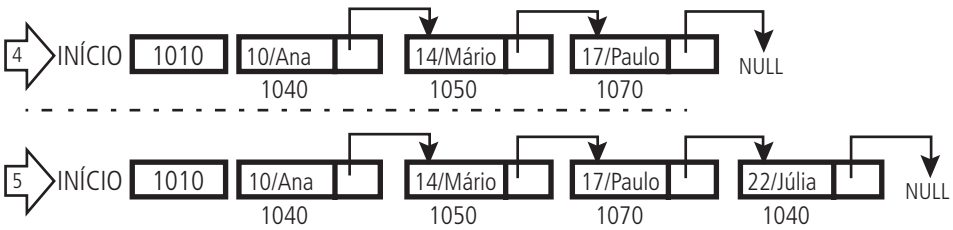


Figura 3.10b: Inserção da lista dinâmica ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Finalmente, no (passo 5), conforme Figura 3.10b tivemos a inserção do aluno com matrícula 22. Esse será inserido no final da lista.

O código abaixo, implementado está em linguagem C, inseri um novo aluno à lista:

```
/* Funcao para inserir um novo no, ao final da lista */
void inserir () {
 TAluno *novono, *antinserido;
 int i, matl,continuar;
 char nomel[15];
 do{
 cabec();
 printf("\nInserir novo aluno \n");
 printf("\nMatricula: ");
 scanf("%d",&matl);
 fflush(stdin);
 printf("\nNome: ");
 gets(nomel);
 qa++;
 //Aloca memoria para o novo aluno e coloca os dados
 novono = (TAluno *) malloc(sizeof(TAluno));
```

```
novono->mat = matl;
for (i=0;i<=14;i++)
novono->nome[i] =nomel[i];
antinserido = NULL;
// Inserir novono na lista de alunos
if (inicio == NULL) {/* Ainda nao existe nenhum aluno na lista */
inicio = novono;
novono->prox = NULL;
}
else {
noatual = inicio;
if (noatual->mat > matl) {// aluno inserido no inicio da lista
novono->prox = inicio;
inicio = novono;
}
else { // aluno sera inserido no meio ou final da lista
while(noatual != NULL) {
if (noatual->mat < matl) {// procura o local da insercao
antinserido = noatual;
noatual = noatual->prox;
}
else // encontrou o local onde sera inserido
noatual = NULL;
}
novono->prox = antinserido->prox;
antinserido->prox = novono;
}
```

```
    }
    printf("\n\nInserido com Sucesso!!!!\n");
    printf("\nContinuar inserindo (1-sim/2-nao)? ");
    scanf("%d",&continuar);
    }while (continuar == 1); // verifica se quer continuar removendo
}
```

II. Consultar elemento

Para fazer uma consulta é necessário primeiro saber qual elemento deseja consultar. Um ponteiro auxiliar será usado para percorrer a lista, visitando cada nó a procura do elemento.

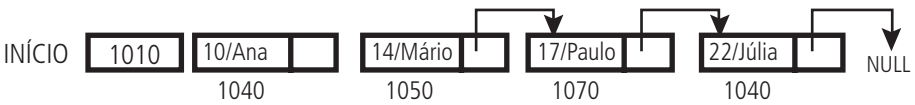


Figura 3.11: Consulta da lista dinâmica ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Na Figura 3.11 temos uma lista com quatro elementos. Caso quiséssemos consultar o elemento de matrícula 25, iríamos percorrer a lista até chegar ao último nó, cujo endereço do vizinho é NULL (nó de endereço 1040) e ficaríamos sabendo que este elemento não se encontra na lista. Se procurássemos a matrícula 8, assim que fosse feita a comparação com o primeiro elemento da lista, já saberíamos que a matricula 8 não se encontra na lista e a consulta é finalizada. A busca é executada enquanto não encontra o elemento procurado ou enquanto não encontra um elemento cuja matrícula é maior do que a matrícula que está sendo procurada.

O código abaixo, implementado está em linguagem C, realiza um consulta de um determinado aluno:

```
/* Consultar um aluno na lista */
void consultar() {
 int matc, continuar, achou=0;
 do{
 cabec();
```

```
printf("\nConsulta aluno pelo numero de matricula\n\n");
printf("Matricula: ");
scanf("%d",&matc);
noatual = inicio; // coloca ponteiro no inicio da lista
while(noatual != NULL) { // percorre a lista procurando o aluno
if (noatual->mat == matc) { // aluno encontrado
achou = 1;
printf("\n\nMat Nome \n");
printf("---------------------------\n");
printf("%2d %-15s\n", noatual->mat, noatual->nome);
printf("---------------------------\n");
break;
}
else {// procura no proximo aluno
noatual = noatual->prox;
if (noatual != NULL) {
if (noatual->mat > matc)
break;
}
}
}
if (achou == 0) // aluno nao esta na lista
printf("\n\nAluno nao encontrado!!\n\n");
printf("\nContinuar consultando (1-sim/2-nao)? ");
scanf("%d",&continuar);
}while (continuar == 1);
}
```

III. Remover elemento

A remoção em uma lista dinâmica ordenada segue a mesma regra de uma lista desordenada. A única diferença é que não teremos o ponteiro fim para atualizar. Para remover um elemento é necessário saber qual elemento deseja remover. Para isso, a matrícula do aluno a ser removido deve ser lida. É feita uma varredura em todos os nós da lista. Assim que o elemento é encontrado, seus dados devem ser apresentados ao usuário (neste caso a matrícula e o nome). Quando o elemento não é encontrado, uma mensagem de erro deve ser dada ao usuário.

Na Figura 3.12 são ilustrados os dois casos de remoção: primeiro da lista e meio ou fim da lista. Quando o primeiro elemento da lista é removido, (passo 2), o endereço do início precisa ser atualizado. No (passo 3) o elemento a ser removido é o último da lista. Neste caso, o elemento que apontava para o elemento removido (1080), terá que apontar para o elemento que o elemento removido aponta (*NULL*).

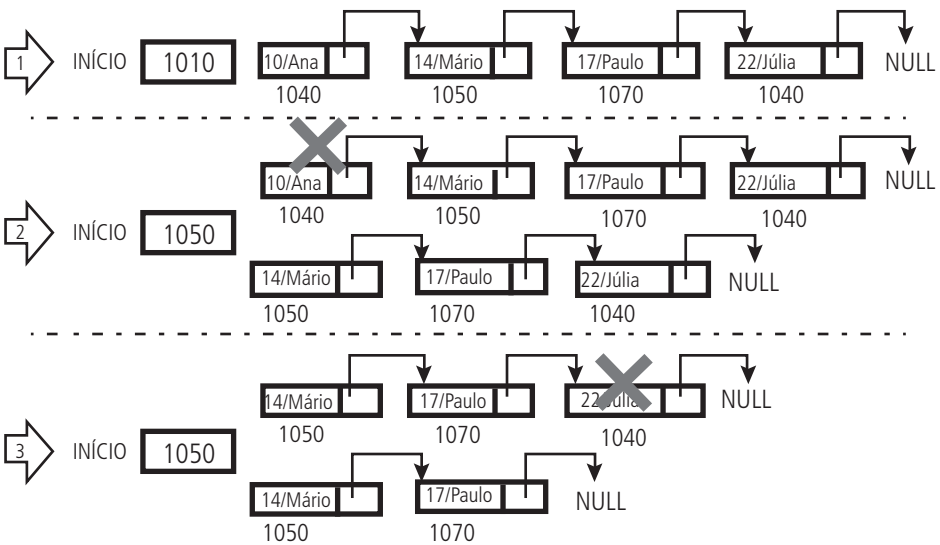


Figura 3.12: Remoção da lista dinâmica ordenada
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código abaixo, implementado está em linguagem C, realiza uma remoção aluno:

```
/* remover um aluno da lista */

void remover() {

 TAluno *noantrem;

 int matr, confrem, continuar, achou;
```

```
do{
achou = 0;
cabec();
printf("\nRemove aluno \n\n");
printf("Matricula: ");
scanf("%d",&matr);
noatual = inicio; //ponteiro que ira percorrer a lista
while(noatual != NULL) { // percorre a lista a procura do aluno
if (noatual->mat == matr) { // verifica se é o aluno
achou = 1;
// impressao dos dados do aluno para conferencia
printf("\n\nMatricula Nome\n");
printf("---------------------------------------\n");
printf("%9d %-15s\n", noatual->mat, noatual->nome);
printf("---------------------------------------\n");
printf("\n\nDeseja remover o aluno (1-sim, 2-nao)? ");
scanf("%d",&confrem);
if (confrem ==1) {// confirma que quer remover
if (noatual == inicio) // verifica se é o primeiro da lista
inicio = inicio->prox;
else // removido esta no meio ou final da lista
noantrem->prox=noatual->prox;
qa--;
free(noatual); // libera memoria do no removido
printf("\n\n Removido com sucesso!!!!\n");
}
```

```
else // cancelou a remocao
printf("\n\n Remocao cancelada\n");
break;
}
else {// passa para o proximo no para continuar a busca
noantrem = noatual;
noatual = noatual->prox;
if (noatual !=NULL) {
if (noatual->mat > matr)
break;
}
}
}
if (achou == 0) // o elemento nao foi encontrado na lista
printf("\n\nAluno nao encontrado!!\n\n");
printf("\n\nDeseja remover outro (1-sim, 2-nao)? ");
scanf("%d",&continuar);
}while (continuar ==1); // continuar removendo aluno
}
```

IV. Listagem de todos os elementos

A operação de listagem possibilita a visualização dos dados de todos os elementos cadastrados. É feita uma varredura na lista partindo do endereço **inicio** até o último nó, todos os dados de todos os elementos são apresentados ao usuário.

O código a seguir representa a impressão de todos os elementos:

```
/* Lista todos os alunos presentes na lista */
void listar () {
 noatual = inicio; // no auxiliar marca o inicio da lista
 cabec();
 printf("\nListagem de Alunos\n\n");
 if (qa != 0) {// verifica se tem aluno cadastrado
 printf("\n\nMatricula Nome\n");
 printf("---------------------------------------\n");
 while( noatual != NULL) { // percorre toda a lista ate chegar no final
 printf("%9d %-15s\n", noatual->mat, noatual->nome);
 noatual = noatual->prox; // Faz noatual apontar para o proximo no
 }
 printf("---------------------------------------\n");
 printf("\n\nQuantidade de Alunos: %d\n",qa);
 }
 else
 printf("\n\nNao tem nenhum aluno cadastrado");
 printf("\n\nTecle enter para voltar para o menu...");
 getche();
}
```

Assista à video-aula da disciplina de Estrutura de Dados disponível no AVEA. Aproveite para revisar os conteúdos da aula 3 sobre listas.

A implementação do código completo da lista dinâmica ordenada na linguagem C estará disponibilizado junto ao AVEA em formatos **.pdf** e .**cpp**

Vamos praticar, através do código completo do conteúdo visto até o momento da aula 3 disponível no AVEA, implemente um novo programa trocando variáveis alunos, matricula e polo por pessoa e endereço.

Resumo

Nesta aula abordamos uma visão geral de estrutura de dados, descrevendo de tipos primitivos e estruturados, descrevemos o tipo Lista (ordenada e desordenada) estática e dinamicamente e implementamos suas operações básicas tais como: inserção, remoção, consulta, listagem na linguagem C.

Atividades de aprendizagem

1. Os tipos de dados são divididos em primitivos e estruturados, comente acerca deles descrevendo-os.

2. Diferencie estruturas homogêneas e heterogêneas.

3. Construa um programa na linguagem C que represente uma Lista Estática Desordenada, que possua como campos: código e nome do cliente. Neste programa será necessário criar um menu que tenha inserir, remover, listar, consulta. Observação: Implemente um contador de registros inseridos e que o mesmo deverá ser decrementado quando a informação foi excluída.

4. Construa um programa na linguagem C que represente uma Lista Dinâmica Desordenada, que possua como campos: código, nome do cliente. Neste programa será necessário criar um menu que tenha inserir, remover, listar, consulta. Observação: Implemente um contador de registros inseridos e que o mesmo deverá ser decrementado quando a informação foi excluída.

Aula 4 – Pilhas

Objetivos

Conhecer o funcionamento de uma Pilha.

Implementar as operações básicas em uma estrutura pilha estática e dinâmica.

4.1. Introdução

Para entendermos o funcionamento de uma estrutura de pilha, podemos fazer uma analogia com uma pilha de pratos. Se quisermos adicionar um prato na pilha, o colocamos no topo. Para pegar um prato da pilha, retiramos o do topo. Assim, temos que retirar o prato do topo para ter acesso ao próximo prato. A estrutura de pilha funciona de maneira análoga. Cada novo elemento é inserido no topo e só temos acesso ao elemento do topo da pilha. Portanto As pilhas são estruturas baseadas no princípio LIFO (*last in, first out*), na qual os dados que foram inseridos por último na pilha serão os primeiros a serem removidos.

Existem três operações básicas que devem ser implementadas numa estrutura de pilha: operação para empilhar um novo elemento, inserindo-o no topo, operação para desempilhar um elemento, removendo-o do topo e a operação para consultar qual elemento está no topo da pilha. É comum nos referirmos a essas operações pelos termos em inglês *push* (empilhar) e *pop* (desempilhar).

4.2. Pilha estática

A seguir vamos analisar a implementação de pilha estática. Neste exemplo será implementada uma pilha de livros.

4.2.1. Operações básicas

I. Inserir elemento

Todo elemento que vai ser inserido em uma pilha é colocado no final da estrutura. A Figura 4.1 ilustra a chegada de três livros colocados na pilha. Os livros vão sendo inseridos por ordem de chegada.

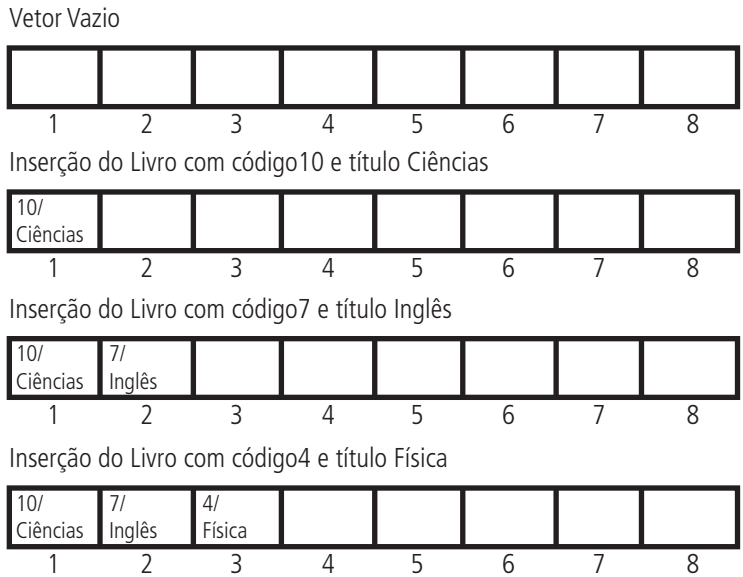


Figura 4.1: Inserção da pilha estática
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código abaixo, implementado está em linguagem C, inseri um novo aluno à pilha:

```
/* Funcao para inserir um novo livro na pilha */
void inserir () {
 int continuar;
 do{
 cabec();
 printf("\nColocar livro no topo da pilha \n");
 printf("\nCodigo do livro: ");
 scanf("%d",&ll.codigo);
 printf("\nTitulo do Livro: ");
 fflush(stdin);
 gets(ll.titulo);
 // Inserir livro na pilha
 if (tampilha <30) { /* Se ainda tem vaga na pilha */
 livro[tampilha] = ll;
 tampilha++;
```

```
 printf("\n\nInserido com Sucesso!!!!\n\n");
 }
 else /* Pilha cheia*/
 printf("\n\nPilha cheia, Livro nao inserido!!!!\n\n");
 printf("\n Continuar inserindo (1-sim/2-nao)? ");
 scanf("%d",&continuar);
 }while (continuar == 1); // verifica se quer continuar inserindo livros
}
```

II. Consultar topo da pilha

Em uma pilha, a consulta é feita apenas do elemento do topo, ou seja, o último elemento a ser inserido. Assim, teremos a informação de qual será o próximo elemento a ser retirado. Na Figura 4.2, o elemento do topo da pilha é o livro de código 4 e título Física, armazenado na posição 3 do vetor. Quando o vetor estiver vazio, é apresentada uma mensagem de pilha vazia ao usuário.

Pilha com três elementos

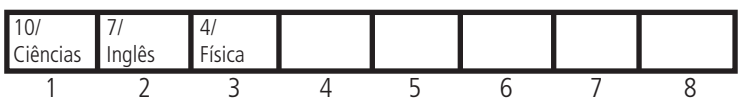


Figura 4.2: Consulta da pilha estática
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código abaixo, implementado está em linguagem C, realiza um consulta de um determinado aluno:

```
/* Consultar topo da pilha*/
void consultatopo() {
 cabec();
 printf("\nConsulta topo da pilha\n");
 if (tampilha != 0) {
 printf("\n\nCod Titulo \n");
 printf("-------------------------------------------------\n");
 printf("%2d %-20s \n", livro[tampilha-1].codigo, livro[tampilha-1].titulo);
```

```
printf("-------------------------------------------------\n");
}
else
printf("\n\nA pilha esta vazia!!\n\n");
printf("\n\nTecle enter para voltar para o menu\n");
getche();
}
```

III. Remover topo da pilha

Em uma pilha, o elemento removido é sempre o que chegou há menos tempo, ou seja, o elemento da última posição do vetor. Na Figura 4.3 iremos remover o elemento do topo da pilha, neste caso, o elemento da posição 3. Não há necessidade de deslocamentos.

Figura 4.3: Remoção da pilha estática
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código abaixo, implementado está em linguagem C, realiza uma remoção aluno:

```
/* remover um livro da pilha */
void retirapilha() {
 int i, confrem, continuar;
 do{ cabec();
 printf("\nRetira livro do topo da pilha \n");
 if (tampilha != 0) {// verifica se tem elementos na pilha
 printf("\n\nCodigo Titulo Editora\n");
 printf("-------------------------------------------------\n");
```

```
 printf("%6d %-20s \n", livro[tampilha-1].codigo, livro[tampilha-1].titulo);
 printf("-------------------------------------------------\n");
 printf("\n\nconfirma retirada do livro (1-sim, 2-nao)? ");
 scanf("%d",&confrem);
 if (confrem ==1) { // confirma que quer remover
 tampilha--;
 printf("\n\n Retirado da Pilha com sucesso!!!!\n\n");
 }
 else // cancelou a remocao
 printf("\n\n Retirada cancelada\n\n");
 }
 else // pilha vazia
 printf("\n\nPilha vazia!!\n\n");
 printf("\n\nDeseja retirar outro livro(1-sim, 2-nao)? ");
 scanf("%d",&continuar);
 }while (continuar ==1); // continuar retirando livro da pilha
}
```

IV. Listagem de todos os elementos da pilha

A operação de listagem possibilita a visualização dos dados de todos os elementos da pilha. É feita uma varredura no vetor e todos os dados de todos os elementos são apresentados ao usuário.

O código abaixo representa a impressão de todos os elementos:

```
/* Lista todos os livros da pilha */
void listar () {
 int i;
 cabec();
```

```
printf("\nListagem dos livros da pilha\n");
if (tampilha != 0) {
printf("\n\nCodigo Titulo Editora\n");
printf("-------------------------------------------------\n");
for (i=tampilha-1;i>=0;i--)
printf("%6d %-20s \n", livro[i].codigo, livro[i].titulo);
printf("-------------------------------------------------\n");
printf("\n\nQuantidade de livros na pilha = %d\n",tampilha);
}
else
printf("\n\n Nao tem nenhum livro na pilha");
printf("\n\n\nTecle enter para voltar para o menu\n");
getche();
}
```

A implementação do código completo da pilha estática na linguagem C estará disponibilizado junto ao AVEA em formatos **.pdf** e **.cpp**!

4.3. Pilha dinâmica

Nesta seção iremos analisar as operações básicas em uma pilha implementada com ponteiros em linguagem C.

4.3.1. Operações básicas

I. Inserir elemento

Todo elemento que vai ser inserido em uma pilha é colocado no final da estrutura. A Figura 4.4 ilustra a chegada de três livros colocados na pilha.

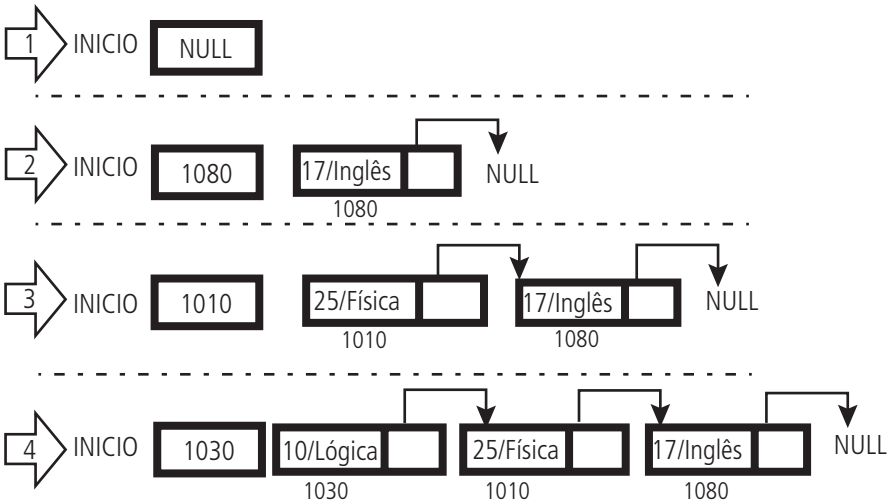


Figura 4.4: Inserção da pilha dinâmica
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Cada elemento que chega, será inserido no início da estrutura. Portando, o ponteiro **inicio** é sempre modificado a cada inserção.

Na Figura 4.4 a pilha inicia vazia, com o ponteiro início apontando para NULL (passo 1). No (passo 2) um elemento é inserido na pilha, por ser o primeiro, ele não tem vizinho e o ponteiro início passa a apontar para o novo nó (endereço 1080).

No (passo 3) um novo elemento é inserido, o ponteiro início deve ser atualizado e o novo nó deve apontar para o elemento que estava no início da pilha (endereço 1080). No (passo 4), um novo elemento é inserido e as atualizações dos ponteiros devem ser feitas.

O código abaixo, implementado está em linguagem C, inseri um novo aluno à pilha:

```
/* Funcao para inserir um novo no, no inicio da pilha */
void inserir () {
 TLivro *novono;
 int i, codl, continuar;
 char titl[30];
 do{
```

```
cabec();
printf("\nColocar livro no topo da pilha \n");
printf("\nCodigo do livro: ");
scanf("%d",&codl);
printf("\nTitulo do Livro: ");
fflush(stdin);
gets(titl);
// Inserir livro na pilha
tampilha++;
//Aloca memoria para o novo livro
novono = (TLivro *) malloc(sizeof(TLivro));
novono->codigo = codl;
for (i=0;i<=29;i++)
novono->titulo[i] =titl[i];
novono->prox = inicio;
inicio = novono;
printf("\n\nInserido com Sucesso!!!!\n\n");
printf("\nContinuar inserindo (1-sim/2-nao)? ");
scanf("%d",&continuar);
}while (continuar == 1); // verifica se quer continuar inserindo livros
}
```

II. Consultar topo da pilha

Em uma pilha, a consulta é feita apenas do elemento do topo, ou seja, o último elemento a ser inserido, que neste caso é o elemento apontado pelo ponteiro **inicio**. Se o ponteiro início aponta para *NULL* significa que a pilha está vazia.

O código a seguir, implementado está em linguagem C, realiza um consulta de um determinado aluno:

```
/* Consultar livro do topo da pilha */
void consultatopo() {
 cabec();
 printf("\nConsulta topo da pilha\n\n");
 if (inicio != NULL) {
 printf("\n\nCodigo Titulo \n");
 printf("-------------------------------------------------\n");
 printf("%6d %-20s \n", inicio->codigo, inicio->titulo);
 printf("-------------------------------------------------\n");
 }
 else
 printf("\nA pilha está vazia!!\n\n");
 printf("\n\nTecle enter para voltar para o menu\n");
 getche();
}
```

III. Remover topo da pilha

Em uma pilha, o elemento removido é sempre o que chegou há menos tempo, ou seja, o elemento apontado pelo ponteiro **inicio**. Na Figura 4.5 iremos remover o elemento do topo da pilha. O ponteiro início deve ser atualizado, apontando para o elemento que era vizinho do elemento removido.

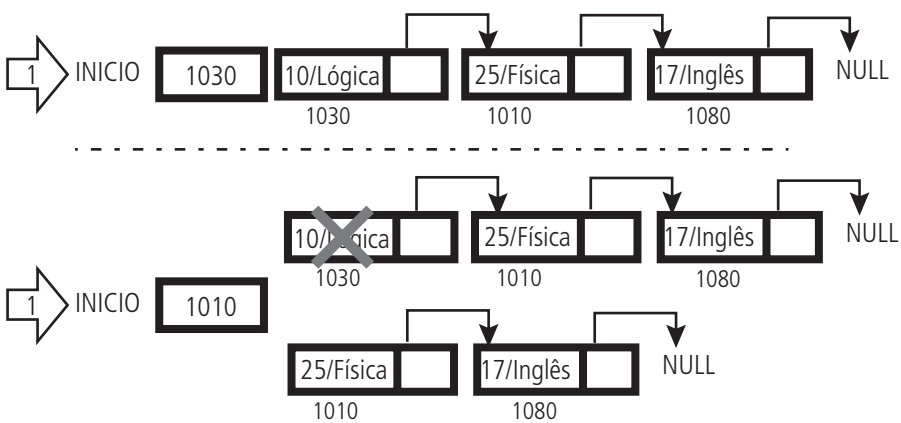


Figura 4.5: Remoção da pilha dinâmica
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

O código abaixo, implementado está em linguagem C, realiza uma remoção aluno:

```
/* remover livro do topo da pilha */
void retirapilha() {
 int confrem, continuar;
 do{
 cabec();
 printf("\nRetira livro do topo da pilha \n");
 noatual = inicio; //ponteiro que ira apontar para o primeiro no
 if (inicio != NULL) { // verifica se tem elementos na pilha
 printf("\n\nCodigo Titulo \n");
 printf("------------------------------------------------\n");
 printf(" %6d %-20s \n", inicio->codigo, inicio->titulo);
 printf("------------------------------------------------\n");
 printf("\n\nconfirma retirada do livro (1-sim, 2-nao)? ");
 scanf(" %d",&confrem);
 if (confrem ==1) { // confirma que quer remover
 inicio = inicio->prox;
 free(noatual); // libera memoria do no removido
 tampilha--;
 printf("\n\n Retirado da Pilha com sucesso!!!!\n\n");
 }
 else // cancelou a remocao
 printf("\n\n Retirada cancelada\n\n");
 }
 else // pilha vazia
```

```
 printf("\n\nPilha vazia!!\n\n");
 printf("\n\nDeseja retirar outro livro(1-sim, 2-nao)? ");
 scanf("%d",&continuar);
 }while (continuar ==1); // continuar retirando livro da pilha
}
```

IV. Listagem de todos os elementos da pilha

A operação de listagem possibilita a visualização dos dados de todos os elementos da pilha. É feita uma varredura na pilha e todos os dados de todos os elementos são apresentados ao usuário.

O código abaixo representa a impressão de todos os elementos:

```
/* Lista todos os livros da pilha */
void listar () {
 noatual = inicio; // no auxiliar marca o inicio da lista
 cabec();
 printf("\nListagem dos livros da pilha\n\n");
 if (inicio != NULL) {
 printf("\n\nCodigo Titulo \n");
 printf("-------------------------------------------------\n");
 while( noatual != NULL) { // percorre toda a pilha
 printf("%6d %-20s \n", noatual->codigo, noatual->titulo);
 noatual = noatual->prox; // Faz noatual apontar para proximo no
 }
 printf("-------------------------------------------------\n");
 printf("\n\nQuantidade de livros na pilha = %d\n",tampilha);
 }
 else
```

```
 printf("\n\n Nao tem nenhum livro na pilha");
 printf("\n\n\nTecle enter para voltar para o menu\n");
 getche();
}
```

A implementação do código completo da pilha dinâmica na linguagem C estará disponibilizado junto ao AVEA em formatos **.pdf** e **.cpp**!

Resumo

Nesta aula descrevemos o tipo pilha estática e dinamicamente e implementamos suas operações básicas tais como: inserção, remoção, consulta, listagem na linguagem C.

Assista à video-aula da disciplina de Estrutura de Dados disponível no AVEA. Aproveite para revisar os conteúdos da aula 3 sobre listas.

Atividades de aprendizagem

1. Qual o critério de inserção e remoção utilizada pela Pilha?

2. A Figura 4.6 representa uma pilha implementada através de um vetor de 10 posições. Observe que são realizadas as operações de remoção e inserção na Pilha. Qual seria a nova configuração da Pilha após as seguintes operações:

   - Remoção;
   - Remoção;
   - Remoção;
   - Inserção do elemento A;
   - Inserção do elemento B;
   - Remoção;
   - Inserção do elemento WW;

Topo da Pilha: [4]

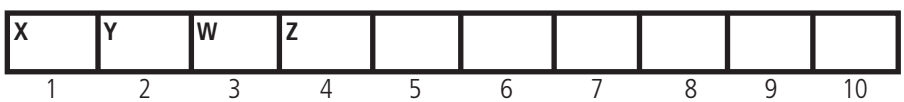


Figura 4.6: Pilha
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

a) Quantos elementos após as remoções e inserções possui o vetor?

b) Quantas posições após as remoções e inserções possui o vetor?

c) Faça o desenho da nova Pilha.

3. Implemente uma Pilha Estática em linguagem C que armazene como dados, 10 (dez) nomes de Pessoas. Implemente as operações de inserção, remoção e consulta.

Observação: colocar o código complemento

4. Implemente o código da questão anterior para Pilha Dinâmica.

Observação: colocar o código complemento

Aula 5 – Filas e árvores

Objetivos

Conhecer e identificar uma Fila Estática e Dinâmica.

Implementar Fila Estática e Dinâmica na linguagem C.

Conceituar Árvore e Árvore Binária; Reconhecer os percursos em árvores binárias.

5.1 Introdução à fila

Para determinadas aplicações é imposto um critério que restringe a inserção e retira dos elementos que compõem um conjunto de dados. O critério escolhido impõe uma ordem ao conjunto de dados, ordem esta que em geral não depende da ordem natural dos valores dos dados.

Um dos critérios mais usuais é:

- Critério FIFO ("*First In First Out*" - Primeiro que entra é o primeiro que sai): dentre os elementos que ainda permanecem no conjunto, o primeiro elemento a ser retirado é o primeiro que tiver sido inserido. Essa estrutura linear com esse tipo de acesso é denominada fila (critério FIFO), e é bastante utilizada na computação e no nosso dia-a-dia. O critério FIFO é o que rege, por exemplo, o funcionamento de uma fila de pessoas em um banco ou de um caixa de supermercado: as pessoas são atendidas na ordem em que chegaram, ou seja, as pessoas que iniciaram a fila serão atendidas primeiro, na frente das que chegaram depois. E quem chegar por último será inserido no final da fila e atendido por último, ou conforme sua posição for mudando. Portanto à medida que a fila anda, de forma sequencial, o primeiro é atendido (retirado), quem era segundo passo a ser primeiro até que chegue ao último da fila. As filas são estruturas de dados que armazenam os elementos de forma sequencial (linear). Elas sofrem inserções e retiradas em extremidades diferentes. Exigem acesso às duas extremidades: começo, onde é feita a retirada, e término, onde é feita a inserção. Portanto, podemos implementar a fila simplesmente

colocando as restrições adequadas nas operações de adicionar e remover elementos de uma lista, parece com o que foi feito com as pilhas, só que agora os elementos são adicionados e removidos de extremidades diferentes.

A pilha segue o critério LIFO (*Last In First Out*), já a fila segue o critério FIFO (*First In, First Out*).

5.1.1 Aplicações de filas no âmbito computacional

São muito comuns, na prática, as aplicações de filas, uma vez que parece bastante natural o critério de atender primeiro (retirar da fila) os elementos que chegaram primeiro. Sendo assim, são exemplos de fila:

- Escalonamento de serviços ("*sob*") submetidos ao Sistema Operacional e colocados em fila para serem executados na ordem em que entraram.

- Sequência de trabalhos submetidos à impressora e colocados em fila para serem impressos.

- Fila de pacotes a serem transmitidos numa rede de comutação de pacotes.

5.1.2 Formas de representação das filas

A implementação de filas se distingue pela natureza dos seus elementos (tipo do dado armazenado), pela maneira como os elementos são armazenados (estática ou dinamicamente) e pelas operações disponíveis para o tratamento da fila. Portanto, uma fila também pode ser representada, de duas formas:

- **Estática:** caracteriza-se por utilizar um vetor (estrutura estática) para representar a fila.

- **Dinâmica:** caracteriza-se por utilizar uma lista encadeada (estrutura dinâmica) para representar a fila.

A implementação dinâmica torna mais simples as operações usando uma lista encadeada com referência para as duas pontas. Já a implementação sequencial é um pouco mais complexa, mas pode ser usada quando há previsão do tamanho máximo da fila. A implementação estática de filas é vantajosa quando há previsão do tamanho máximo de fila. Porém, se a quantidade máxima de elementos de uma fila não for conhecida a priori, a implementação estática torna-se desvantajosa, podendo resultar numa alocação de

memória grande demais ou insuficiente para uma determinada aplicação. Já as filas dinâmicas são bastante vantajosas quando não se conhece o tamanho máximo da fila. Dessa forma, à medida que desejamos inserir novos elementos na fila, basta alocar um espaço de memória adequado e inserir este novo elemento. Porém, como em listas encadeadas, a principal desvantagem da implementação dinâmica está no fato de que um elemento da fila sempre conterá uma referência para o outro elemento da fila, o que gera uma utilização maior de memória.

5.2 Fila estática

Nesta seção iremos verificar as operações em uma fila implementada com vetores. E sua implementação em C.

As filas podem usar vetor, ou seja, uma estrutura estática para armazenar os dados, como pode ser constatado na Figura 5.1 a seguir. Um vetor de 10 (dez) posições, dos quais estão sendo utilizadas 4 posições. No início da Fila, posição 1 tem-se o elemento A e no final da Fila, posição 4 tem-se o elemento D. Observe que a Fila possui 4 elementos, mas o vetor suporta 10, ou seja, existem 6 posições vazias.

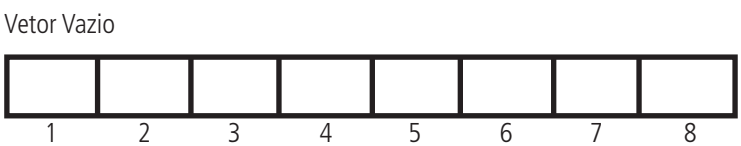


Figura 5.1: Fila estática
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

As operações básicas que podem ser implementadas em uma fila são:

I. criar uma fila vazia;

II. inserir elemento no final da fila;

III. remover o elemento do início da fila;

IV. consultar o primeiro da fila;

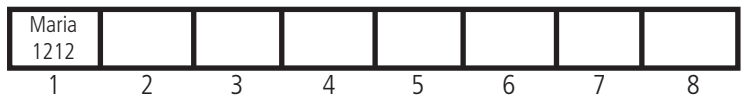
V. listar todos os elementos da fila.

Para um melhor entendimento do que seja uma fila, temos abaixo uma implementação de uma SIMULAÇÃO de fila de matrículas no curso técnico de informática.

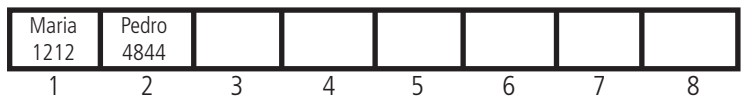
a) Fila vazia

No início o vetor está vazio, ou seja, fila vazia;

Inserção da aluna de matrícula 1212 e nome Maria



Inserção da aluna de matrícula 4844 e nome Pedro



Inserção da aluna de matrícula 5611 e nome José

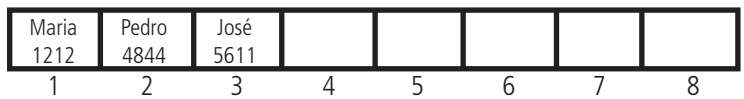


Figura 5.2: Fila estática vazia
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

b) Inserir elemento

Todo elemento que vai ser inserido em uma fila é colocado no final da estrutura. Os alunos vão sendo inseridos por ordem de chegada. No exemplo da Figura 5.3 são inseridos 3 alunos na fila.

Fila com 3 alunos

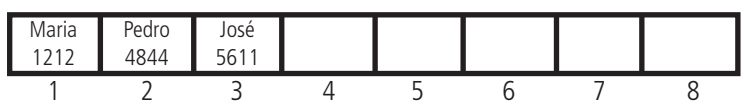


Figura 5.3: Fila estática inserção
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Logo abaixo, temos a função inserir ( ) implementada na linguagem C para inserir um novo aluno no final da fila.

```
void inserir () {
        int continuar;
        do {
                printf("\n Chegada de novo aluno na fila \n");
                 printf("\n Numero da Matricula: ");
```

```
                scanf("%d",&Al.Matricula);
                printf("\n Nome: ");
                fflush(stdin);
                gets(Al.Nome);
                printf("\n Polo do Aluno(1- Batalha, 2-Valenca): ");
                scanf("%d",&Al.Polo);
                if (tamfila <30) { /* Se ainda tem vaga na fila */
                        Aluno[tamfila] = Al;
                        tamfila++;
                        printf("\n\nAluno Inserido com Sucesso!!!!\n\n");
                }
                else /* Fila cheia*/
                        printf("\n\nFila cheia, Aluno nao inserido!!!!\n\n");
                printf("\n Continuar inserindo (1-sim/2-nao)? ");
                scanf("%d",&continuar);
        }while (continuar == 1);
}
```

c) Consultar primeiro da fila

Em uma fila, a consulta é feita apenas do primeiro elemento da fila. Assim, teremos a informação de qual será o próximo elemento a ser retirado. Na Figura 5.4, o primeiro elemento da fila é a aluna Maria, com matrícula 1212.

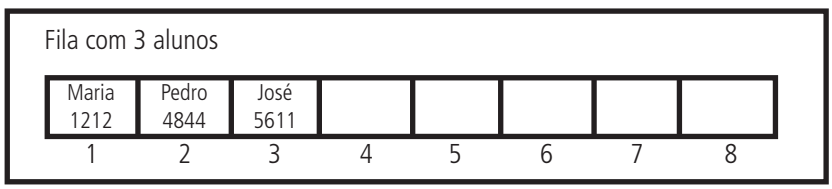


Figura 5.4: Fila Estática Consulta
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Logo abaixo, temos a função **consultarprimeiro( )** implementada na linguagem C para consultar o primeiro aluno da fila.

```
void consultarprimeiro ( ) {
        cabec();
        printf("\nConsulta primeiro aluno da fila\n");
        if (tamfila != 0) {
                printf("\nMatricula Nome Polo\n");
                printf("-----------------------------------------\n");
                printf("%4d %-15s %2d\n",
                Aluno[0].Matricula, Aluno[0].Nome, Aluno[0].Polo);
                printf("-----------------------------------------\n");
        }
        else
                printf("\n\nA fila está vazia!!\n\n");
        printf("\n\nTecle enter para voltar para o menu\n");
        getche();
}
```

d) Remover primeiro da fila

Observando a Figura 5.5, em uma fila, o elemento removido é sempre o que chegou há mais tempo, ou seja, o elemento da primeira posição do vetor. Na figura abaixo iremos remover o primeiro elemento da fila (neste caso, Maria-1212). Os elementos seguintes devem ser deslocados uma posição a frente. Com isso, o segundo elemento da fila passa a ser o primeiro, o terceiro passa a ser o segundo e assim por diante.

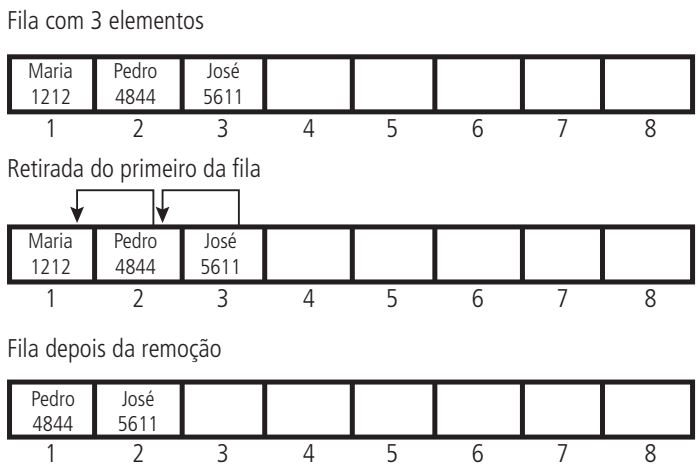


Figura 5.5: Fila estática remoção
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Logo abaixo, temos a função **retirafila ( )** implementada na linguagem C para remover o aluno da fila.

```
void retirafila ( ) {
        int i, confrem, continuar;
        do{
                printf("\nRetira primeiro aluno da fila \n");
                if (tamfila != 0) {
                        printf("\n\nMatricula Nome Polo\n");
                        printf("---------------------------------------\n");
                        printf("%4d %-15s %2d\n", Aluno[0].Matricula,
                        Aluno[0].Nome, Aluno[0].Polo);
                        printf("-------------------------------------------------\n");
                        printf("\n\n Confirma retirada do aluno (1-sim,
2-nao)? ");
                        scanf("%d",&confrem);
                        if (confrem ==1) {
                                for (i=0; i<tamfila; i++)
                                        Aluno[i] = Aluno[i+1];
                                tamfila--;
                                printf("\n\n Aluno retirado da fila com
                                sucesso!!!!\n\n");
                        }
                        else
                                printf("\n\n Retirada cancelada\n\n");
                }
                else
                        printf("\n\nFila vazia!!\n\n");
                printf("\n\nDeseja retirar outro aluno(1-sim, 2-nao)? ");
                scanf("%d",&continuar);
        }while (continuar ==1);
}
```

e) Listagem de todos os elementos da fila

A operação de listagem possibilita a visualização dos dados de todos os elementos da fila. É feita uma varredura no vetor e todos os dados de todos os elementos são apresentados.

Logo abaixo, temos a função **listar ( )** implementada na linguagem C para remover o aluno da fila.

```
void listar () {
        int i;
        cabec ( );
        printf("\nListagem de alunos da fila\n");
        if (tamfila != 0) {
                printf("\nMatricula Nome Polo\n");
                printf("----------------------------------------\n");
                for (i=0;i<tamfila;i++)
                        printf("%4d %-15s %2d\n",
                        Aluno[i].Matricula, Aluno[i].Nome, Aluno[i].Polo);
                printf("----------------------------------------\n");
                printf("\n\nQuantidade de alunos na fila = %d\n",tamfila);
        }
        else
                printf("\n\n Nao tem nenhum aluno na fila");
        printf("\n\n\nTecle enter para voltar para o menu\n");
        getche ( );
}
```

A implementação do código completo da fila estática na linguagem C estará disponibilizado junto ao AVEA em formatos **.pdf** e **.cpp!**

Depois de vermos como funcionam as operações básicas em uma fila estática (vetor) e sua implementação na linguagem C, vamos entender o que é uma FILA DINÂMICA e vê sua implementação, na próxima seção.

5.3 Fila dinâmica

Nesta seção iremos verificar as operações em uma fila implementada com ponteiros, por meio de uma lista. O ponteiro da lista, que aponta para o primeiro elemento, representa o início da fila (ponteiro I). Outro ponteiro para o último elemento da lista representa a parte final da fila (ponteiro F). Seja a fila Q, implementada por meio de uma lista, temos a Figura 5.6 a seguir. E posteriormente sua implementação em C.

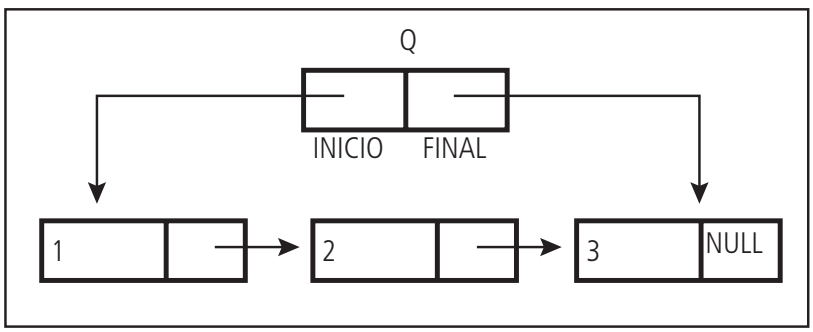


Figura 5.6: Fila dinâmica
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

A Fila dinâmica é sempre implementada com 2 ponteiros, o 1° (primeiro) no início da fila e o 2° (segundo) no final.

As operações básicas que são implementadas em uma fila estática são as mesmas na fila dinâmica. A implementação abaixo também SIMULA fila de matrículas no curso técnico de informática.

I. Inserir elemento

Todo elemento que vai ser inserido em uma fila é colocado no final da estrutura.

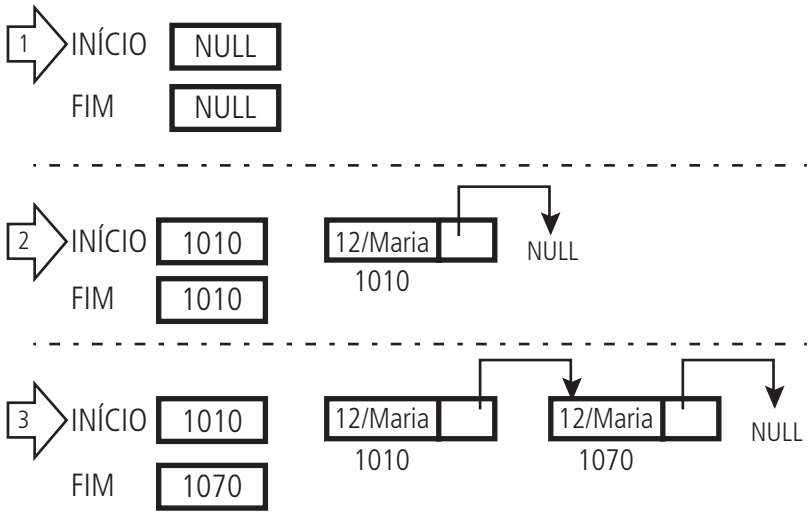


Figura 5.7: Fila dinâmica – inserção
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

A Figura 5.7 ilustra a chegada de dois alunos na fila. Inicialmente a fila está vazia, portanto o valor do ponteiro início e fim é *NULL* (passo 1). Quando um elemento vai ser inserido, a memória é alocada e os dados são armazenados (passo 2). Todo nó criado em uma fila dinâmica, não tem vizinho seguinte, por isso ele aponta para *NULL*. Na inserção do primeiro elemento, os ponteiros início e fim apontam para este elemento (no exemplo, endereço de memória 1010). No passo 3 temos a chegada da aluna Ana-10, que será armazenado no final da estrutura, o último elemento da fila o terá (endereço de memória 1070) como vizinho e o ponteiro fim irá apontar para o novo elemento.

Logo abaixo, temos a função **inserir ( )** implementada na linguagem C para inserir um novo aluno no final da fila dinâmica, agora utilizando ponteiros.

```
void inserir () {
        TAluno *novono;
        int i, Matricula1, Pol1, continuar;
        char Nome1[15];
        do{
            printf("\n Chegada de novo aluno na fila \n");
            printf("\n Numero da Matricula: ");
```

```
                scanf("%d",&Matriculal);
                printf("\n Nome: ");
                fflush(stdin);
                gets(Nomel);
                printf("\n Polo do Aluno(1- Batalha, 2- Valenca): ");
                scanf("%d",&Polol);
                tamfila++;
                novono = (TAluno *) malloc(sizeof(TAluno));
                novono->Matricula = Matriculal;
                for (i=0;i<=14;i++)
                        novono->Nome[i] =Nomel[i];
                novono->Polo = Polol;
                novono->prox = NULL;
                        // Inserir novo Aluno na fila de Alunos
                if (inicio == NULL) {
                        inicio = novono;
                        fim = novono;
                }
                else {
                        fim->prox = novono;
                        fim = novono;
                }
                printf("\n\nAluno Inserido com Sucesso!!!!\n\n");
                printf("\n Continuar inserindo (1-sim/2-nao)? ");
                scanf("%d",&continuar);
        }while (continuar == 1); // verifica se quer continuar inserindo
}
```

II. Consultar primeiro da fila

Em uma fila, a consulta é feita apenas do primeiro elemento da fila. Assim teremos a informação de qual será o próximo elemento a ser retirado. O primeiro elemento da fila é o elemento do endereço início. Quando início estiver apontando para *NULL*, significa que a fila está vazia.

Logo abaixo, temos a função **consultarprimeiro ( )** implementada na linguagem C para consultar o primeiro aluno da fila dinâmica.

```
void consultarprimeiro ( ) {
        printf("\nConsulta primeiro aluno da fila\n\n");
        noatual = inicio; // coloca ponteiro no inicio da lista
        if (noatual != NULL) {
                printf("\n\n Matricula Nome Polo\n");
                printf("-----------------------------------------------\n");
                printf("%4d %-15s %2d\n", noatual->Matricula,
                noatual->Nome, noatual->Polo);
                printf("-----------------------------------------------\n");
        }
        else
                printf("\nA fila está vazia!!\n\n");
        printf("\n\nTecle enter para voltar para o menu\n");
        getche();
}
```

III. Remover primeiro da fila

Na Figura 5.8, em uma fila, o elemento removido é sempre o que chegou há mais tempo, ou seja, o elemento apontado por início. Quando um elemento é removido, o endereço do ponteiro início deve apontar para o vizinho do primeiro da fila.

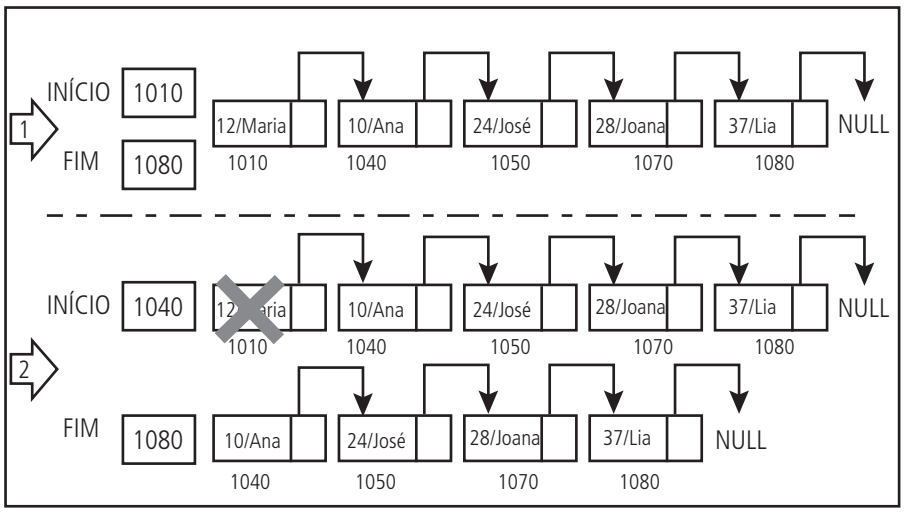


Figura 5.8: Fila dinâmica – remoção
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Logo abaixo, temos a função **retirafila ( )** implementada na linguagem C para retirar o primeiro aluno da fila dinâmica.

```
void retirafila() {
        int confrem, continuar;
        do{
                printf("\nRetira primeiro Aluno da fila \n\n");
                noatual = inicio;
                if (noatual != NULL) {
                        printf("\n\nMatricula Nome Polo\n");
                        printf("----------------------------------------\n");
                        printf("%4d %-15s %2d\n", noatual->Matricula,
                        noatual->Nome, noatual->Polo);
                        printf("----------------------------------------\n");
                        printf("\n\nconfirma retirada do aluno (1-sim,
2-nao)? ");
                        scanf("%d",&confrem);
```

```
                        if (confrem ==1) {
                                inicio = inicio->prox;
                                free(noatual);
                                tamfila--;
                                printf("\n\n Retirado da fila com sucesso!!!!\
n\n");
                        }
                        else
                                printf("\n\n Retirada cancelada\n\n");
                }
                else            // fila vazia
                        printf("\n\nFila vazia!!\n\n");
                printf("\n\nDeseja retirar outro Aluno(1-sim, 2-nao)? ");
                scanf("%d",&continuar);
        }while (continuar ==1);         // continuar retirando cliente da
fila
}
```

IV. Listagem de todos os elementos da fila

A operação de listagem possibilita a visualização dos dados de todos os elementos da fila. É feita uma varredura na fila e todos os dados de todos os elementos são apresentados.

Logo abaixo, temos a função **listar( )** implementada na linguagem C para mostrar todos os alunos matriculados, ou seja, inseridos na fila dinâmica.

```
void listar () {
        noatual = inicio;
        cabec();
        printf("\nListagem de Aluno da fila\n\n");
        if (tamfila != 0) {
                printf("\nMatricula Nome Polo\n");
                printf("----------------------------------------\n");
                while( noatual != NULL) {
                        printf("%4d %-15s %2d\n", noatual->Matricula,
                        noatual->Nome, noatual->Polo);
                        noatual = noatual->prox;
                }
                printf("------------------------------------------------\n");
                printf("\n\nQuantidade de Aluno na fila = %d\n",tamfila);
        }
        else
                printf("\n\n Nao tem nenhum aluno na fila");
        printf("\n\n\nTecle enter para voltar para o menu\n");
        getche();
}
```

Assista à video-aula "Estrutura de Dados - Aula 05 - Parte I" disponível em http://etapi.cefetpi.br/Videos/Estrutura_de_dados_5_parte1/index.html. Aproveite para revisar os conteúdos da aula sobre filas e árvores.

A implementação do código completo da fila dinâmica na linguagem C estará disponibilizado junto a plataforma AVEA em formatos **.pdf** e **.cpp**!

5.4 Introdução à árvore

Além do estudo de representações de dados lineares (lista, filas e pilhas), este unidade trata das chamadas genericamente de não-lineares (árvores). As árvores permitem que sejam feitos outros tipos de relações entre os dados, nesse caso os dados (denominados nós ou nodos) são subordinados uns aos outros

Na estrutura de dados árvore existe uma hierarquia entre o conjunto de dados pertencentes a mesma.

Um exemplo bem conhecido de relação de estruturação em árvore é a estrutura de uma "universidade" composta de "centros". Cada "centro" composto por um certo número de "departamentos". Cada "departamento" oferece um conjunto de "disciplinas", nas quais estão matriculados os "alunos".

A Figura 5.9 representa um esquema desta hierarquia é mostrado abaixo:

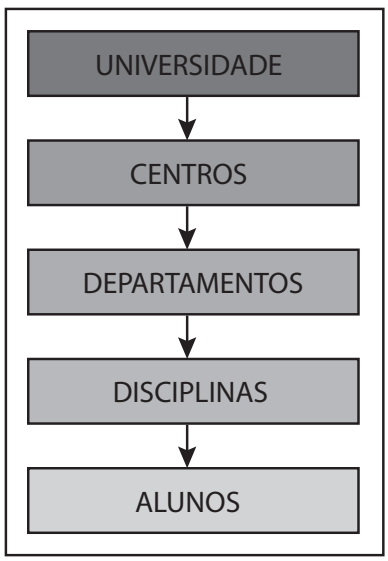


Figura 5.9: Organograma de uma universidade
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Outros exemplos de estruturas em forma de árvores são:

- A divisão de um livro em capítulos, seções, tópicos.
- A árvore genealógica de uma pessoa.
- A organização dos diretórios (pastas) do Sistema Operacional *Windows Vista*.

Formalmente, uma árvore é um conjunto finito T de um ou mais nós, tais que:

a) existe um nó denominado raiz da árvore.

b) os demais nós formam m≥ 0 conjuntos disjuntos S_1, S_2, ..., Sm onde cada um destes conjuntos é uma árvore. As árvores Si ($1 \leq i \leq n$) recebem a denominação de subárvores.

Uma árvore nunca poderá ter mais de uma raiz.

Em computação, árvores (e especialmente árvores binárias) são usadas para armazenar dados (chaves e outros campos de informação) em seus nós da mesma forma que listas ligadas e vetores. E assim como as demais estruturas de dados, as árvores também possuem sua representação gráfica. As três formas mais comuns de representação gráfica de uma árvore são demonstradas a seguir:

1. Representação por parênteses aninhados
(A (B) (C (D (G) (H)) (E) (F (I))))

2. Diagrama de inclusão

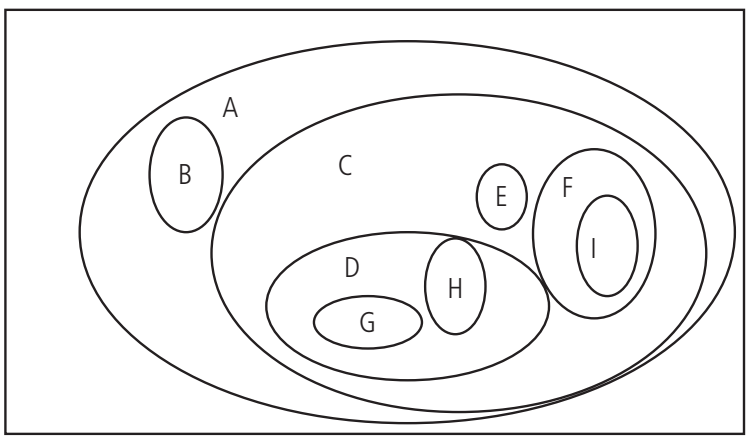


Figura 5.10: Organograma de uma universidade
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

3. Representação hierárquica

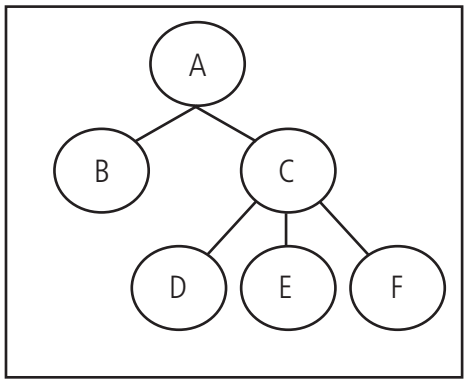


Figura 5.11: Árvore cheia de grau 2
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

A representação hierárquica da árvore é a mais conhecida no meio computacional.

5.5 Árvore binária

Uma vez adquirido o conhecimento a respeito de árvore, apresentaremos agora os conceitos sobre árvore binária. Uma árvore binária é formada por um conjunto finito de nós. Este conjunto ou é vazio ou consiste de um nó raiz com duas árvores binárias disjuntas, denominadas subárvores da esquerda e da direita. Pela definição de árvore, cada nó da árvore é a raiz de uma subárvore. O número de subárvores de um nó é o grau daquele nó. Um nó de grau igual a zero é denominado folha ou nó terminal. O grau de cada nó é menor ou igual a dois. Nas árvores binárias, distinguem-se as subárvores de um nó entre subárvores da esquerda e subárvores da direita. Assim, se o grau de um nó for igual a 1, deve ser especificado se a sua subárvore é a da esquerda ou a da direita. Uma árvore binária também pode ser vazia, isto é, não possuir nenhum nó. É importante observar que uma árvore binária não é apenas uma árvore de grau máximo dois, pois há também a questão de ordem (esquerda e direita) de subárvores, conceito este que não existe na definição de árvore comum discutida no item anterior. A Figura 5.12 representa um exemplo de árvore binária de nível 3.

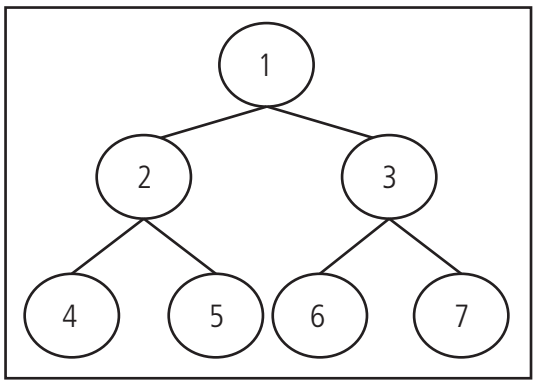


Figura 5.12: Árvore binária nível 3
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Se, por exemplo, n1 for a raiz de uma árvore binária e n2 for a raiz da sua sub-árvore direita ou esquerda, então diz-se que n1 é o pai de n2, e que n2 é o filho direito ou esquerdo de n1. O nó n2 é um descendente de n1 e n1 é o seu ascendente. Um nó que não tenha filhos é uma folha. Dois nós são irmãos se são os filhos direito e esquerdo do mesmo pai. Nível de um nó numa árvore binária: a raiz da árvore tem nível 1. O nível de outro nó qualquer é uma unidade mais que o nível do pai. Uma árvore binária completa de nível n é uma árvore em que cada nó de nível "n" é uma folha e em que todos os nós de nível menor que "n" têm sub-árvores direita e esquerda não vazias. Exemplificando melhor, temos a árvore da Figura 5.13 a seguir.

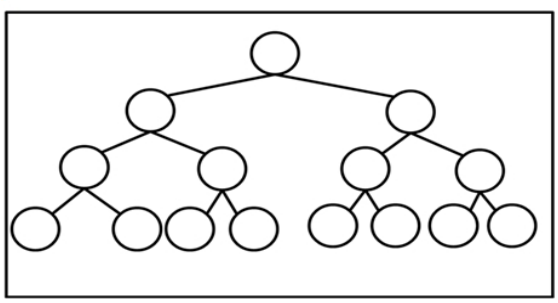


Figura 5.13: Árvore binária
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Podemos concluir que:

- Essa árvore possui 9 nós distribuídos aleatoriamente. O nó A é a raiz da árvore, que tem 2 sub-árvores com B e C, como raízes.

- O número de sub-árvores de um nó determina o grau desse nó. Dessa forma, A e C tem grau 2 e B tem grau 1. O nós que tem grau zero são denominados terminais ou folhas, nesse caso o nó F.

- Utilizando a relação de hierarquia existente na árvore binária, podemos conhecer os antepassados de um nó, basta identificarmos todos os nós ao longo do caminho entre a raiz e este nó. Ex: os antepassados de G são, na ordem: os nós A-B-D.

- Outro conceito importante no estudo de árvores e o conceito de nível, que representa a distância do nodo até a raiz. Por definição, a raiz da árvore tem nível 1. Os nós B e C têm nível 2, os nós D, E e F têm nível 3, e assim por diante. O nó de maior nível nos fornece a altura (ou profundidade) da árvore. Sendo assim, a árvore acima apresenta altura 4.

Podemos também calcular o número máximo de nós possíveis em uma árvore binária, pode-se utilizar o valor da altura h e a fórmula **2h – 1**, como está exemplificado na Figura 5.14 a seguir.

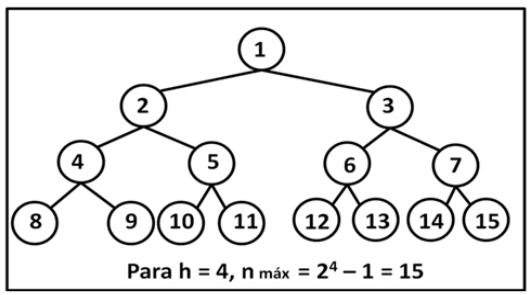


Figura 5.14: Cálculo de nós da árvore
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Para uma melhor exemplificação de árvores binárias pode ser utilizar a representação de expressões aritméticas, de tal forma que a hierarquia (prioridade) dos operadores fique clara (bem definida). A representação é feita de tal forma que a prioridade das operações fique implícita: o operador de menor prioridade da expressão aritmética aparece na raiz da árvore; a subexpressão que forma o operando da esquerda deste operador dá origem à subárvore da esquerda da raiz; enquanto que a subexpressão que forma o operando que forma o operando da direita dá origem à subárvore da direita da raiz. Os operandos são sempre representados nos nós terminais, conforme exemplificada na Figura 5.15. Suponhamos as seguintes expressões aritméticas:

(3 + 6) * (4 - 1) + 5

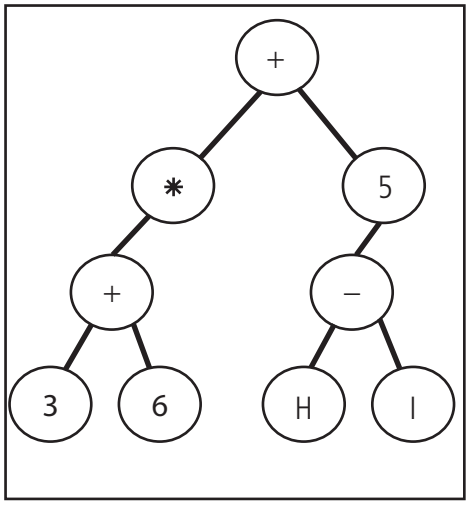


Figura 5.15: Árvore binária de expressão aritmética
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Observe que se a expressão possuir parênteses, estes não aparecerão na árvore.

5.5.1 Percurso em árvores binárias

Há diversas formas de manipulação de árvores binárias e, em geral, estas supõem o exame dos conteúdos (informações) dos nós. O acesso sistemático aos nós de uma árvore de maneira que cada nó seja examinado no máximo uma única vez (para que não haja repetição), sugere que a árvore seja percorrida segundo algum critério pré-estabelecido. Esta ação de percorrer a árvore, com a condição de que cada nó seja examinado no máximo uma vez, denomina-se caminhamento na árvore binária. Se for realizado um caminhamento numa árvore de forma que todos os seus nós sejam visitados (acessados), os mesmos são implicitamente organizados segundo uma ordem linear. Dependendo do critério estabelecido para caminhar na árvore, obtém-se uma sequência dos nós correspondentes $\mathbf{x_{i1}, x_{i2}, \ldots x_{in}}$ em que:

- **n** é o número de nós da árvore;
- $\mathbf{x_j}$ é o conteúdo do nó que ocorre na j-ésima posição na sequência de caminhamento da árvore; e
- $\mathbf{x_{ik}}$ ocorre antes de $\mathbf{x_{ip}}$ na sequência se o nó com informação **xik** é visitado antes do que o nó $\mathbf{x_{ip}}$, segundo o caminhamento escolhido.

Em geral são utilizadas três formas de caminhamentos em árvores binárias e estas são determinadas dependendo da ordem em que são visitados o nó raiz, sua subárvore à esquerda e sua subárvore à direita, o termo "visitar" significa a realização de alguma operação sobre a informação do nó, como modificação da mesma, impressão ou qualquer outra. No caso de árvores binárias, existem determinadas ordens de caminhamento mais frequentemente utilizadas. As três ordens principais são:

a) Caminhamento pré-fixado (raiz-esquerda-direita):

1. visita a raiz;
2. percorre a subárvore da esquerda;
3. percorre a subárvore da direita.

b) Caminhamento in-fixado (esquerda-raiz-direita):

1. percorre a subárvore da esquerda;
2. visita a raiz;
3. percorre a subárvore da direita.

c) Caminhamento pós-fixado (esquerda-direita-raiz):

1. percorre a subárvore da esquerda;
2. percorre a subárvore da direita;
3. visita a raiz.

O termo visita está sendo usado para indicar o acesso a um nó para fins executar alguma operação sobre ele.

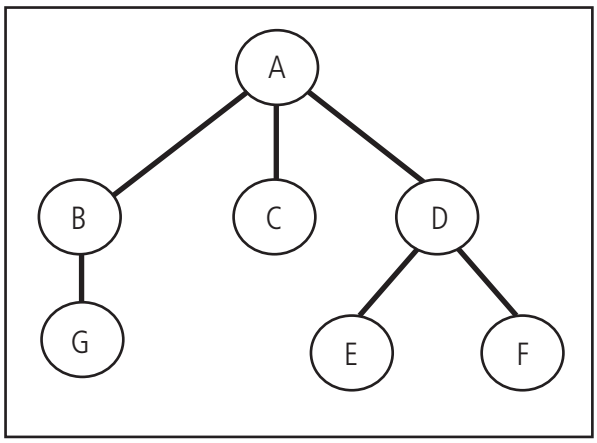


Figura 5.16: Árvore binária
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

As três ordens de caminhamento acima apresentadas, aplicadas sobre a árvore da Figura 5.16, produzem as seguintes sequências:

- pré-fixado: A B G C D E F;
- in-fixado: G B C E F D A;
- pós-fixado: G F E D C B A.

Assista à video-aula "Estrutura de Dados - Aula 05 - Parte II" disponível em http://etapi.cefetpi.br/Videos/Estrutura_de_dados_5_parte2/index.html. Aproveite para revisar os conteúdos da aula sobre **filas e árvores**..

Aula 6 – Ordenação e pesquisa

Objetivos

Classificar ou ordenar dados.

Implementar o método da Bolha ou BubbleSort.

Conceituar e implementar diferentes estratégias de pesquisa.

6.1 Introdução à ordenação

Classificação ou ordenação de dados constitui uma das tarefas mais frequentes e importantes em processamento de dados, sendo, normalmente, auxiliar ou preparatória, visando a tornar mais simples e eficientes as demais. Em diversas aplicações, os dados devem ser armazenados obedecendo uma determinada ordem. Alguns algoritmos podem explorar a ordenação dos dados para operar de maneira mais eficiente, do ponto de vista de desempenho computacional. Para obtermos os dados ordenados, temos basicamente duas alternativas: ou inserimos os elementos na estrutura de dados respeitando a ordenação (dizemos que a ordenação é garantida por construção), ou, a partir de um conjunto de dados já criado, aplicamos um algoritmo para ordenar seus elementos. Os métodos de ordenação são ser empregados em aplicações computacionais. Devido ao seu uso muito frequente, é importante ter à disposição algoritmos de ordenação ***sorting*** eficientes tanto em termos de tempo (devem ser rápidos) como em termos de espaço (devem ocupar pouca memória durante a execução). Vamos descrever os algoritmos de ordenação considerando o seguinte cenário:

A-Z

Sort
Classificar, ordenar (em inglês).

- a entrada é um vetor cujos elementos precisam ser ordenados;
- a saída é o mesmo vetor com seus elementos na ordem especificada;
- o espaço que pode ser utilizado é apenas o espaço do próprio vetor.

A importância da classificação de dados pode ser avaliada se considerarmos o que seria o problema da localização do nome de um assinante em uma lista telefônica, na qual os nomes não estivessem em ordem alfabética.

Na verdade, ordenação de dados é o processo pelo qual é determinada a ordem na qual devem se apresentar as entradas de uma tabela de modo que obedeçam à sequência citada por um ou mais de seus campos. Estes campos especificados como determinantes da ordem são chamados chaves de ordenação.

Portanto, vamos discutir ordenação de vetores. Como veremos, os algoritmos de ordenação podem ser aplicados a qualquer informação, desde que exista uma ordem definida entre os elementos. Podemos, por exemplo, ordenar um vetor de valores inteiros, adotando uma ordem crescente ou decrescente.

Podemos também aplicar algoritmos de ordenação em vetores que guardam informações mais complexas, por exemplo um vetor que guarda os dados relativos a alunos de uma turma, com nome, número de matrícula, etc. Nesse caso, a ordem entre os elementos tem que ser definida usando uma das informações do aluno como chave da ordenação: alunos ordenados pelo nome, alunos ordenados pelo número de matrícula, etc. Nos casos em que a informação é complexa, raramente se encontra toda a informação relevante sobre os elementos do vetor no próprio vetor; em vez disso, cada componente do vetor pode conter apenas um ponteiro para a informação propriamente dita, que pode ficar em outra posição na memória. Assim, a ordenação pode ser feita sem necessidade de mover grandes quantidades de informação, para re-arrumar as componentes do vetor na sua ordem correta. Para trocar a ordem entre dois elementos, apenas os ponteiros são trocados. Em muitos casos, devido ao grande volume, a informação pode ficar em um arquivo de disco, e o elemento do vetor ser apenas uma referência para a posição da informação nesse arquivo.

Na próxima seção, será descrito a analisado o método de ordenação por troca, conhecido como Bolha ou *BubbleSort*. Por troca, entende-se os métodos baseados na troca de posição dos dados, de forma a ordená-los. Estes métodos caracterizam-se por efetuarem a ordenação sucessiva de pares de elementos, trocando-os de posição caso estejam fora da ordem desejada.

6.2 Método da bolha ou *bubblesort*

Esse método é muito simples de implementar, ele efetua a ordenação por comparação sucessiva de pares de elementos, trocando-os de posição caso estejam fora da ordem desejada, se necessário, a troca de dois elementos adjacentes, e procura levar os valores mais altos (ou mais baixos) para o final da sequência a ser ordenada.

O algoritmo de "Ordenação Bolha", ou "*Bubblesort*", recebeu este nome pela imagem pitoresca usada para descrevê-lo: os elementos maiores são mais leves, e sobem como bolhas até suas posições corretas.

A ideia fundamental é fazer uma série de comparações entre os elementos do vetor. Quando dois elementos estão fora de ordem, há uma inversão e esses dois elementos são trocados de posição, ficando em ordem correta. Assim, o primeiro elemento é comparado com o segundo. Se uma inversão for encontrada, a troca é feita. Em seguida, independente se houve ou não troca após a primeira comparação, o segundo elemento é comparado com o terceiro, e, caso uma inversão seja encontrada, a troca é feita. O processo continua até que o penúltimo elemento seja comparado com o último. Com este processo, garante-se que o elemento de maior valor do vetor será levado para a última posição. A ordenação continua, posicionando o segundo maior elemento, o terceiro, etc., até que todo o vetor esteja ordenado.

A Figura 6.1 mostra o que acontece com o vetor no decorrer da execução do método da bolha.

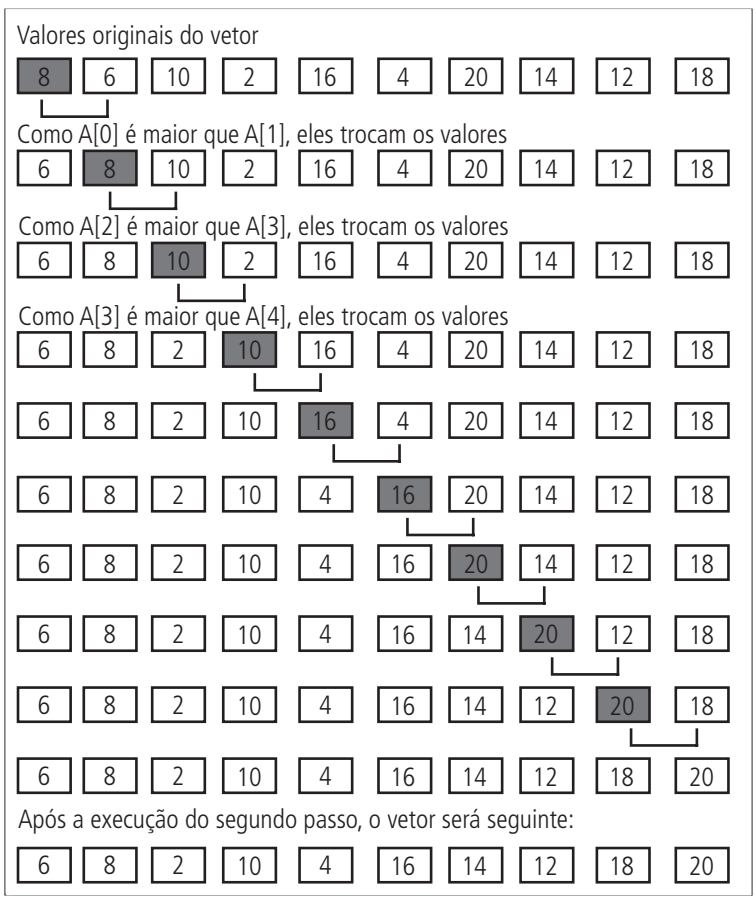


Figura 6.1: Execução do método da bolha
Fonte: FRANÇA S. V. A. Apostila de Estrutura de Dados

Iniciaremos comparando os dois primeiros elementos (o primeiro com o segundo). Como eles estão desordenados, então, trocamos as suas posições (8 e 6).

Comparamos agora o segundo com o terceiro (8 e 10). Como eles estão desordenados, então, trocamos as suas posições, e assim sucessivamente. Chegamos agora ao final do vetor! Note que após essa primeira fase de comparações e trocas o maior elemento, 20, está na sua posição correta e final. Na próxima fase, todo o processo será repetido, porém com a diferença que agora as comparações e trocas não serão mais feitas até o final do vetor, mais sim até o último número que antecede aquele que chegou a sua posição correta no nosso caso o penúltimo elemento. Vamos mais uma vez iniciar o processo, comparando os dois primeiros elementos do vetor. Verificamos que quando os valores estão ordenados, não será necessária a troca de posições. Note que as comparações e trocas dessa fase chegaram ao final, uma vez que o último elemento já está em sua posição correta e que levamos o segundo maior elemento para a sua posição correta. Faz-se isso até que o vetor esteja totalmente ordenado.

Uma função que implementa esse algoritmo na linguagem C é apresentada a seguir. A função recebe como parâmetros o número de elementos e o ponteiro do primeiro elemento do vetor que se deseja ordenar. Vamos considerar a ordenação de um vetor de valores inteiros.

```
/* Algoritmo de Ordenação Bolha */
void bolha (int n, int* v) {
        int i, j;
        for (i=n-1; i>=1; i--) {
                for (j=0; j<i; j++) {
                        if (v[j] > v[j+1]) {          /* troca */
                                int temp = v[j];
                                v[j] = v[j + 1];
                                v[j+1] = temp;
                        }
                }
        }
}
```

Embora o algoritmo de ordenação da Bolha ou *BubbleSort* seja simples, este pode ser altamente ineficiente e é raramente usado do ponto de vista computacional.

Abaixo a função principal para testar esse algoritmo, lembrando que a função *getch ( )* é responsável por ficar aguardando que uma tecla seja pressionada para que o programa continue sua execução normal.

```
/* Função Principal */
/* Testa Algoritmo de Ordenação Bolha */
int main (void) {
        int i;
        int v[8] = { 25, 48, 37, 12, 57, 86, 33, 92};
        printf("Vetor Desordenado: \n");
        for (i=0; i<8; i++)
                printf("%d ",v[i]);
        printf("\n");
        printf("\nPressione qualquer tecla para visualizar o Vetor
        Ordenado\n\n");
        getch();
        bolha(8,v);
        printf("Vetor Ordenado: \n");
        for (i=0; i<8; i++)
                printf("%d ",v[i]);
        printf("\n\n");
        printf("Pressione qualquer tecla para finalizar o programa\n\n");
        getch();
        return 0;
}
```

Para evitar que o processo continue mesmo depois de o vetor estar ordenado, podemos interromper o processo quando houver uma passagem inteira sem trocas, usando uma variante do algoritmo apresentado acima:

```
/* Ordenação Bolha 2ª Versão */
void bolha2 (int n, int* v) {
        int i, j;
        for (i=n-1; i>0; i--) {
                int troca = 0;
                for (j=0; j<i; j++)
                        if (v[j]>v[j+1]) {  /* troca */
                                int temp = v[j];
                                v[j] = v[j+1];
                                v[j+1] = temp;
                                troca = 1;
                        }
                        if (troca == 0) /* nao houve troca */
                                return;
        }
}
```

Implemente um código em C, que utilize um vetor de 20 elementos inteiros, gerado com valores lidos pelo teclado. E ordene esse algoritmo utilizando o método da Bolha. Poste no AVEA da disciplina o arquivo de sua atividade.

Esse mesmo algoritmo pode ser aplicado a vetores que guardam outras informações. O código escrito acima pode ser reaproveitado, a menos de alguns detalhes. Primeiro, a assinatura da função deve ser alterada, pois deixamos de ter um vetor de inteiros; segundo, a forma de comparação entre os elementos também deve ser alterada Para aumentar o potencial de reuso do nosso código, podemos reescrever o algoritmo de ordenação apresentado acima tornando-o independente da informação armazenada no vetor.

A implementação do código completo em linguagem C que demonstra o método de ordenação bolha numa lista desordenada e estática de alunos relacionados pela matrícula, nome, notas e média. estará disponibilizado no AVEA em formatos .**pdf** e **.cpp**!

6.3 Introdução à pesquisa

Finalizando nosso caderno didático, discutiremos diferentes estratégias para efetuarmos a pesquisa de um elemento num determinado conjunto de dados. A operação de pesquisa é encontrada com muita frequência em aplicações computacionais, sendo portanto importante estudar estratégias distintas para efetuá-la. Por exemplo, um programa de controle de estoque pode pesquisar, dado um código numérico ou um nome, a descrição e as características de um determinado produto. Se temos um grande número de produtos cadastrados, o método para efetuar a pesquisa deve ser eficiente, caso contrário a pesquisa pode ser muito demorada, inviabilizando sua utilização. Conforme veremos, certos métodos de organizar dados tornam o processo de busca mais eficiente. Como a operação de busca é uma tarefa muito comum em computação, o conhecimento desses métodos é de fundamental importância para quem vai lidar com programação.

Assista à video-aula da disciplina de Estrutura de Dados – Aula 06 – Parte I disponível em http://cefetpi.nucleoead.net/etapi/. Aproveite para revisar os conteúdos da aula sobre **ordenação e pesquisa.**

Inicialmente, consideraremos que temos nossos dados armazenados em um vetor e discutiremos os algoritmos de pesquisa que podemos empregar. Portanto, dado um vetor *vet* com *n* elementos, desejamos saber se um determinado elemento elem está ou não presente no vetor. Estudaremos algumas estratégias de pesquisa que são:

- **Pesquisa Sequencial**: consiste basicamente em pesquisar num vetor consiste em percorrermos o vetor, elemento a elemento, verificando se o elemento de interesse é igual a um dos elementos do vetor.
- **Pesquisa Binária**: resumidamente é pesquisar o elemento que buscamos com o valor do elemento armazenado no meio do vetor com dados ordenados.

A seguir teremos um melhor discernimento sobre as estratégias de pesquisa.

Identifique situações do dia-a-dia que podem ser facilitadas através da utilização dos métodos de Pesquisa. Poste no AVEA da disciplina o arquivo de sua atividade.

6.4 Pesquisa sequencial

A pesquisa sequencial é um método extremamente simples, mas pode ser muito ineficiente quando o número de elementos no vetor for muito grande. Isto porque o algoritmo (a função, no caso) pode ter que procura do primeiro ao último elemento do vetor até encontrar ou não um determinado

elemento. Portanto, a forma mais simples de fazermos uma pesquisa num vetor consiste em percorrermos o vetor, elemento a elemento, verificando se o elemento de interesse é igual a um dos elementos do vetor. Esse algoritmo pode ser implementado conforme ilustrado pelo código a seguir, considerando-se um vetor de números inteiros. A função apresentada tem como valor de retorno o índice do vetor no qual foi encontrado o elemento; se o elemento não for encontrado, o valor de retorno é –1.

```
int pesquisa (int n, int* vet, int elem) {
        int i;
        for (i=0; i<n; i++) {
                if (elem == vet[i])
                return i; /* elemento encontrado */
        }
/* percorreu todo o vetor e não encontrou elemento */
        return -1;
}
```

Esse algoritmo de pesquisa é extremamente simples, mas pode ser muito ineficiente quando o número de elementos no vetor for muito grande. Isto porque o algoritmo (a função, no caso) pode ter que percorrer todos os elementos do vetor para verificar que um determinado elemento está ou não presente. Dizemos que no pior caso será necessário realizar n comparações, onde n representa o número de elementos no vetor. Portanto, o desempenho computacional desse algoritmo varia linearmente com relação ao tamanho do problema.

Abaixo temos a função principal para testar esse algoritmo de pesquisa linear. Lembrando que a função *getch( )* é responsável por ficar aguardando que uma tecla seja pressionada para que o programa continue sua execução normal.

```
#include <stdio.h>
#include <conio.h>
/* Programa que faz a pesquisa em um vetor */
int main (void) {
```

```
        int v[8] = {24,11,47, 38,58,77,66,82};
        int e = 58; /* informação que se deseja pesquisar */
        int p;
        p = pesquisa(8,v,e);
        if (p == -1)
                printf("Elemento nao encontrado.\n");
        else
                printf("Elemento encontrado no indice:%d ", p);
        getch ();
        return 0;
}
```

Em diversas aplicações reais, precisamos de algoritmos de pesquisa mais eficientes. Seria possível melhorarmos a eficiência do algoritmo de pesquisa mostrado acima? Infelizmente, se os elementos estiverem armazenados em uma ordem aleatória no vetor, não temos como melhorar o algoritmo de pesquisa, pois precisamos verificar todos os elementos. No entanto, se assumirmos, por exemplo, que os elementos estão armazenados em ordem crescente, podemos concluir que um elemento não está presente no vetor se acharmos um elemento maior, pois se o elemento que estamos pesquisando estivesse presente ele precederia um elemento maior na ordem do vetor.

O código abaixo ilustra a implementação da pesquisa linear assumindo que os elementos do vetor estão ordenados (vamos assumir ordem crescente).

```
int pesquisa_ord (int n, int* vet, int elem) {
        int i;
        for (i=0; i<n; i++) {
                if (elem == vet[i])
                        return i; /* elemento encontrado */
                else
                        if (elem < vet[i])
```

```
                    return -1; /* interrompe pesquisa */
            }
            /* percorreu todo o vetor e não encontrou elemento */
            return -1;
}
```

6.4 Pesquisa binária

No caso dos elementos do vetor estarem em ordem, podemos aplicar um algoritmo mais eficiente para realizarmos a pesquisa.

Trata-se do algoritmo de pesquisa binária. A ideia do algoritmo é testar o elemento que buscamos com o valor do elemento armazenado no meio do vetor com dados ordenados. Muito parecido com o método que usamos quando desejamos procurar um número numa lista telefônica. Se o elemento que estamos pesquisando for menor que o elemento do meio, sabemos que, se o elemento estiver presente no vetor, ele estará na primeira parte do vetor; se for maior, estará na segunda parte do vetor; se for igual, achamos o elemento no vetor. Se concluirmos que o elemento está numa das partes do vetor, repetimos o procedimento considerando apenas a parte que restou: comparamos o elemento que estamos pesquisando com o elemento armazenado no meio dessa parte. Este procedimento é continuamente repetido, subdividindo a parte de interesse, até encontrarmos o elemento ou chegarmos a uma parte do vetor com tamanho zero.

Antes de realizar uma pesquisa binária é necessário ordenar seus dados.

Utilizando o método de pesquisa binária, podemos ter por exemplo, um vetor de tamanho fixo, com elementos dispostos em ordem crescente e possuindo por exemplo, 100 elementos. Ao invés de iniciar a pesquisa no elemento inicial (0) e se entender até o elemento final (100), esse método testa o elemento médio (50) e verificar se o número a ser encontrado é maior ou menor que o valor contido naquele índice. Se for menor, como o vetor está em ordem crescente, podemos desprezar todos os números acima de 50 e recomeçar a pesquisa no intervalo de 0 a 50. Esse procedimento deverá se repetir até que o número correspondente seja encontrado ou não exista no vetor. Um procedimento muito mais eficiente do que a pesquisa sequencial.

O código a seguir ilustra uma implementação de pesquisa binária num vetor de valores inteiros ordenados de forma crescente.

```
int pesquisa_bin (int n, int* vet, int elem) {
        /* no inicio consideramos todo o vetor */
        int ini = 0;
        int fim = n-1;
        int meio;
        /* enquanto a parte restante for maior que zero */
        while (ini <= fim) {
                meio = (ini + fim) / 2;
                if (elem < vet[meio])
                        fim = meio – 1; /* ajusta posicão final */
                else
                        if (elem > vet[meio])
                                ini = meio + 1; /* ajusta posicão inicial */
                        else
                                return meio; /* elemento encontrado */
        }
        /* não encontrou: restou parte de tamanho zero */
        return -1;
}
```

O desempenho desse algoritmo é muito superior ao de pesquisa linear. Novamente, o pior caso caracteriza-se pela situação do elemento que estamos procurando não estar no vetor. Quantas vezes precisamos repetir o procedimento de subdivisão para concluirmos que o elemento não está presente no vetor? A cada repetição, a parte considerada na pesquisa é dividida à metade.

O algoritmo de pesquisa binária consiste em repetirmos o mesmo procedimento recursivamente, podendo ser naturalmente implementado de forma recursiva. Embora a implementação não recursiva seja mais eficiente e mais adequada para esse algoritmo, a implementação recursiva é mais sucinta e vale a pena ser apresentada. Na implementação recursiva, temos dois casos a serem tratados. No primeiro, a pesquisa deve continuar na primeira metade do vetor, logo chamamos a função recursivamente passando como parâmetros o número de elementos dessa primeira parte restante e o mesmo ponteiro para o primeiro elemento, pois a primeira parte tem o mesmo primeiro elemento do que o vetor como um todo. No segundo caso, a pesquisa deve continuar apenas na segunda parte do vetor, logo passamos na chamada recursiva, além do número de elementos restantes, um ponteiro para o primeiro elemento dessa segunda parte. Para simplificar, a função de pesquisa apenas informa se o elemento pertence ou não ao vetor, tendo como valor de retorno falso (0) ou verdadeiro (1). Uma possível implementação usando essa estratégia é mostrada a seguir.

```
int pesquisa_bin_rec (int n, int* vet, int elem) {
 /* testa condição de contorno: parte com tamanho zero */
        if (n <= 0)
                return 0;
        else {
                /* deve pesquisar o elemento entre os índices 0 e n - 1 */
                int meio = (n - 1) / 2;
                if (elem < vet[meio])
                        return pesquisa_bin_rec(meio,vet,elem);
                else
                         if (elem > vet[meio])
                                return pesquisa_bin_rec(n – 1-meio,
                                &vet[meio + 1],elem);
                        else
                                return 1; /* elemento encontrado */
        }
}
```

Em particular, devemos notar a expressão *&vet[meio+1]* que, como sabemos, resulta num ponteiro para o primeiro elemento da segunda parte do vetor. Se quisermos que a função tenha como valor de retorno o índice do elemento, devemos acertar o valor retornado pela chamada recursiva na segunda parte do vetor.

Os mesmos algoritmos de pesquisa sequencial e binária podem ser aplicados a vetores que guardam outras informações, além de números inteiros.

A implementação do código completo em linguagem C, onde o código representa um sistema de cadastro de alunos num curso técnico de informática, onde são armazenadas as informações referente à matrícula, nome, situação (aprovado ou reprovado) de cada aluno, e ainda duas notas e a partir dessas notas é gerado uma média. O código para aplicar a pesquisa sequencial e binária estará disponibilizado no AVEA em formatos **.pdf** e **.cpp**!

Resumo

Nesta aula abordamos conceitos de classificação ou ordenação de dados. O método de ordenação adotado foi o método de ordenação por troca, conhecido como Bolha ou BubbleSort. Por se tratar de um método muito simples de implementar, recebeu este nome pela imagem pitoresca usada para descrevê-lo: os elementos maiores são mais leves, e sobem como bolhas até suas posições corretas. E foi feita a implementação do BubbleSort na linguagem C. Após ordenarmos os elementos de um determinado conjunto, necessário se faz saber realizar uma pesquisa. Foram então, disponibilizadas informações sobre estratégias de pesquisa conhecidas como: Pesquisa Sequencial e Binária. E suas posteriores implementações na linguagem C.

Assista à video-aula da disciplina de Estrutura de Dados – Aula 06 – Parte II disponível em http://cefetpi.nucleoead.net/etapi/. Aproveite para revisar os conteúdos da aula sobre ordenação e pesquisa.

Atividades de aprendizagem

1. Demonstre graficamente a ordenação dos dados 25, 48, 37, 12, 57, 86, 33, 92 usando o método da Bolha.

2. Implemente um código em C, que utilize um vetor de 2000 elementos inteiros, gerado com valores lidos pelo teclado. E ordene esse algoritmo utilizando o método da Bolha.

3. Em diversas aplicações reais, precisamos de algoritmos de busca mais eficientes. Seria possível melhorarmos a eficiência do algoritmo de busca?

Referências

ASCENCIO, A. F. G., CAMPOS, E. A. V. **Fundamentos da Programação de Computadores. Editora Pearson Prentice Hall**. 2ª edição, 2007.

FRANÇA S. V. A. **Apostila de Estrutura de Dados**. Acessado em 04/12/2011 http://www.4shared.com/office/MWxzbSi9/Estrutura_de_Dados_-_Sonia_Vir.html.

HOROWITZ, E.; SAHNI, S. **Fundamentos de Estruturas de Dados.** Rio de Janeiro, Ed. Campus, 1986.

LEISERSON, C., RIVEST, L. **Algoritmos - Teoria e Pratica**. Ed.Campus, 1ª edição, 2002.

MIZRAHI, Victorine Viviane. **Treinamento em C**. Ed Pearson Prentice Hall, 2ª edição, 2008.

PINTO, Wilson Silva. **Introdução ao Desenvolvimento de Algoritmos e Estrutura de dados.** Ed. Érica LTDA.

TENEMBAUM, Aaron M. **Estrutura de Dados Usando C.** São Paulo: Makron Books do Brasil, 1995.

TREMBLAY, Jean Paul e RICHARD, B. Bunt. **Ciências dos Computadores, Uma Abordagem Algorítmica.** Ed. McGraw Hill.

SCHILDT, H. **C Completo e Total**, Ed. Makron Books, 3ª edição, 1997.

SZWARCFITER, J Luiz; MARKENZON, Lilian. **Estruturas de Dados e seus Algoritmos**. Rio de Janeiro: Livros Técnicos e Científicos Editora S. A, 2 ª edição, 1994.

VELLOSO, Paulo. **Estruturas de Dados.** Rio de Janeiro: Ed. Campus, 1991.

VILLAS, Marcos V & Outros. **Estruturas de Dados:** Conceitos e Técnicas de implementação. Rio de Janeiro: Ed. Campus, 1993.

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

Currículo dos professores-autores

Nádia Mendes dos Santos é bacharel em Ciências Contábeis pela Universidade Federal do Piauí, Campus Ministro Reis Velloso, em Parnaíba, e Tecnólogo em Processamento de Dados pela Universidade Estadual do Piauí, Campus de Parnaíba. Tem pós-graduação "*Lato Sensu*" em Banco de Dados, pelo Centro Federal de Educação do Piauí. Mestre em Engenharia Elétrica pela Universidade Federal do Ceará. E atualmente faz Doutorado em Computação pela Universidade Federal Fluminense, linha de pesquisa em Inteligência Artificial. Desde 2006, é professora do quadro efetivo do Instituto Federal de Ciência e Tecnologia do Piauí, Campus Angical do Piauí, na categoria de Dedicação Exclusiva. Seus principais interesses de pesquisa e atuação são inteligência artificial, estrutura de dados, eletrônica digital, microprocessadores digitais e linguagens de programação. E faz parte da Equipe de Educação à Distância do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, atuando como tutora à distância do polo de Monsenhor Gil.

Geraldo Nunes da Silva Junior é bacharel em Ciência da Computação, licenciado em Letras/Inglês e pós-graduado "*Lato Sensu*" em Docência do Ensino Superior pela Universidade Estadual do Piauí. Professor do quadro efetivo do Instituto Federal de Ciência e Tecnologia do Piauí desde 2009, Campus Piripiri. Professor de informática do quadro efetivo da rede pública municipal de Educação. Foi professor da Universidade Estadual do Piauí por 7 anos, lecionando disciplinas: Lógica de Programação, Linguagens de Programação, Estrutura de Dados, Banco de Dados. Presente junto à Equipe de Educação à Distância do Instituto Federal de Educação, Ciência e Tecnologia do Piauí desde seu início, atuando como professor regente e agora professor conteudista.

Otílio Paulo da Silva Neto é bacharel em Sistemas de Informação pela Faculdade Integral Diferencial. Tem pós-graduação "*Lato Sensu*" em Docência do Ensino Superior. E atualmente é diretor - Netsoft Tecnologia Ltda, Coordenador de Tecnologia do Núcleo de Educação a Distância do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, professor substituto da Universidade Estadual do Piauí, professor titular do Instituto Federal de Educação, Ciência e Tecnologia do Piauí. Tem experiência na área de Ciência da Computação, com ênfase em Banco de Dados e Programação para *Web*, atuando principalmente nos seguintes temas: Netsac – Sistema de Gestão Comercial e Industrial, patrimônio, procedimentos médicos, plano de saúde e *NetProcess* - Sistema de Gestão de processos totalmente via *web*.

rede e-Tec Brasil

ISBN 978-85-67082-02-8