TÉCNICAS DE PROGRAMAÇÃO

AUTORES
Fábio José Parreira
Guilherme Bernardino da Cunha
Teresinha Letícia da Silva
Adriana Soares Pereira

LICENCIATURA EM COMPUTAÇÃO

TÉCNICAS DE PROGRAMAÇÃO

AUTORES
Fábio José Parreira
Guilherme Bernardino da Cunha
Teresinha Letícia da Silva
Adriana Soares Pereira

1ª Edição
UAB/NTE/UFSM

UNIVERSIDADE FEDERAL DE SANTA MARIA

Santa Maria | RS
2019

Este caderno foi elaborado pelo Núcleo de Tecnologia Educacional da Universidade Federal de Santa Maria para os cursos da UAB.

NÚCLEO DE TECNOLOGIA EDUCACIONAL

DIRETOR DO NTE
Paulo Roberto Colusso

ELABORAÇÃO DO CONTEÚDO
Fábio José Parreira, Guilherme Bernardino da Cunha, Teresinha Letícia da Silva e Adriana Soares Pereira.

REVISÃO LINGUÍSTICA
Camila Marchesan Cargnelutti
Maurício Sena

APOIO PEDAGÓGICO
Carmen Eloísa Berlote Brenner
Keila de Oliveira Urrutia

EQUIPE DE DESIGN
Carlo Pozzobon de Moraes
Juliana Facco Segalla – Diagramação
Matheus Tanuri Pascotini
Reginaldo Júnior – Diagramação
Raquel Bottino Pivetta
Lisiane Dutra Lopes

PROJETO GRÁFICO
Ana Letícia Oliveira do Amaral

T255 Técnicas de programação [recurso eletrônico] / Fábio José Parreira ...[et al.]. – 1. ed. – Santa Maria, RS : UFSM, NTE, 2019.
1 e-book : il.

Este caderno foi elaborado pelo Núcleo de Tecnologia Educacional da Universidade Federal de Santa Maria para os cursos da UAB
Acima do título: Licenciatura em Computação
ISBN 978-85-8341-256-4

1. Informática 2. Programação - técnicas I. Parreira, Fábio José II. Universidade Federal de Santa Maria. Núcleo de Tecnologia Educacional

CDU 004.42

Ficha catalográfica elaborada por Lizandra Veleda Arabidian - CRB-10/1492
Biblioteca Central da UFSM

MINISTÉRIO DA EDUCAÇÃO
PÁTRIA AMADA BRASIL GOVERNO FEDERAL

CAPES
UAB UNIVERSIDADE ABERTA DO BRASIL
Universidade Federal de Santa Maria 1960
PROGRAD UFSM
NÚCLEO DE TECNOLOGIA EDUCACIONAL

APRESENTAÇÃO

A informática, desde o seu surgimento, vem evoluindo a passos galopantes. Além disso, ela é considerada a ciência da informação, pois é responsável por gerir toda e qualquer informação digitalizada. De acordo com esta premissa é correto afirmar que de nada adiantaria a evolução, tanto do hardware quanto do software, se não houvesse a evolução na forma de armazenamento das informações. Podemos entender que as estruturas de dados, nada mais são, do que, o estudo da otimização de armazenamento e tratamento das informações.

Logo, este estudo é o cerne para qualquer curso de informática. Neste contexto, esse e-book trata das principais técnicas da área de Estruturas de Dados. Logo, é um assunto muito importante para o seu desenvolvimento profissional, pois irá lhe proporcionar conhecimentos para que você decida qual a melhor forma de manipular, em seus futuros programas de computador, os dados computacionais.

As estruturas de dados, na sua maioria dos casos, foram inspiradas em formas naturais de armazenamento do nosso dia-a-dia, como por exemplo, as filas e pilhas. Cada uma destas estruturas possui a sua aplicação ótima, logo possuem vantagens e desvantagens.

Para que você possa entender e implementá-las, este livro foi dividido em 6 unidades:

- Unidade 1: Introdução à programação - apresenta os conceitos iniciais sobre estruturas de dados, apresenta os tipos abstratos de dados (TAD), aritmética de ponteiros, alocação de memória e passagem de parâmetros por referência;
- Unidade 2: Listas lineares - apresenta o estudo e implementação de listas encadeadas e duplamente encadeadas;
- Unidade 3: Pilhas e filas - apresenta as teorias e implementações das pilhas e filas usando TAD;
- Unidade 4: Árvore - apresenta os conceitos sobre árvores, focando nas árvores binárias, e implementa uma árvore binária de busca usando TAD;
- Unidade 5: Algoritmos de Ordenação - apresenta e implementa os três principais algoritmos de ordenação, o *insertion sort, buble sort* e o *quick sort;*
- Unidade 6: Tabela *Hash* - apresenta as funções *hash* mais importantes e as formas de tratamento de colisões e implementa uma tabela *hash* como exemplo de algoritmo de busca.

E por fim, cabe ressaltar que em todos os assuntos abordados, nesse livro, a base para entendê-los são os ponteiros e listas. Cabe destacar que a pilha e fila são listas com disciplina de acesso, árvore é uma lista não linear e a tabela *hash* é implementada usando listas. Desta forma, se você aprender a programar as estruturas de listas, certamente não terá dificuldades nas demais estruturas, pois o que mudará nos demais conteúdos é a forma de inserir os blocos de memória alocados e a forma de acessar os dados.

Veja que o estudo sobre Técnicas de Programação não é difícil, este conteúdo exige que seja construído o alicerce bem sedimentado, nos dois conteúdos supracitados (ponteiros e listas). Construa o seu alicerce e siga em frente.

Bons estudos.

ENTENDA OS ÍCONES

ATENÇÃO: faz uma chamada ao leitor sobre um assunto, abordado no texto, que merece destaque pela relevância.

INTERATIVIDADE: aponta recursos disponíveis na internet (sites, vídeos, jogos, artigos, objetos de aprendizagem) que auxiliam na compreensão do conteúdo da disciplina.

SAIBA MAIS: traz sugestões de conhecimentos relacionados ao tema abordado, facilitando a aprendizagem do aluno.

TERMO DO GLOSSÁRIO: indica definição mais detalhada de um termo, palavra ou expressão utilizada no texto.

SUMÁRIO

1

INTRODUÇÃO À PROGRAMAÇÃO

INTRODUÇÃO

Ao estudarmos a história dos computadores, percebe-se que desde a sua criação, o principal objetivo é o de facilitar e agilizar os processamentos complexos. Para satisfazer tal objetivo, o computador recebe e armazena grandes volumes de dados, submete-os a processamentos e ao final emite resultados. O nosso problema inicia ao receber tais dados. Sendo assim, como devemos escolher as informações corretas, acerca do problema a ser resolvido, de tal forma que essas informações representem adequadamente a questão no mundo computacional?

A esse processo damos o nome de abstração, que é o ato de representar conceitualmente os dados do mundo real, no mundo computacional. Assim, as informações inseridas no computador, consistem em um conjunto cuidadosamente selecionado de dados do objeto real. Como devemos fazer seleção das informações, certos dados do mundo real são desprezados, por serem inexpressivos para a solução do problema. Desta forma, abstrair pode ser entendido como uma simplificação dos dados.

Sendo assim, toda vez que vamos resolver um problema com o auxílio do computador, é necessário definir a abstração da realidade. Destaca-se que esta atividade não é tarefa fácil, mas ela deve ser orientada às características do problema a ser resolvido. Por exemplo, temos que construir um programa que calcule o salário dos funcionários ao final do mês, sendo assim, no modelo abstrato, os dados relevantes para este problema são: o nome dos funcionários; dias trabalhados; o valor do salário ao mês. Outras informações como a altura, cor do cabelo e peso são irrelevantes para o problema em questão, portanto, não deverão ser selecionadas.

Ao definirmos o conjunto de dados que devem ser inseridos no computador, temos que definir agora como ele será armazenado por meio do tipo de dado atribuído a cada elemento do conjunto abstraído.

Neste panorama, esta unidade apresenta alguns conceitos referentes ao armazenamento da informação no computador, que é a base para iniciar o estudo sobre técnicas de programação. Logo, estudaremos nesta unidade os tipos de dados, alocação de espaço em memória secundária e como se trabalha com ponteiros. Lembrem-se que este conteúdo faz parte dos pilares desta disciplina.

SAIBA MAIS:

O que é mesmo Abstração?

Uma vez que a abstração é fundamental para programação de computadores, procure saber mais sobre este assunto, busque outras fontes e definições.

1.1
TIPOS DE DADOS

Para cada dado armazenado no computador, temos que definir o conjunto de valores que ele pode assumir, a esse conjunto denominamos de tipo de dados. Por exemplo, um dado (variável) do tipo lógico pode assumir o valor verdadeiro (true) ou falso (false).

Portanto, a definição dos tipos pode sofrer variações de acordo com a linguagem escolhida. Neste e-book vamos construir os nossos protótipos em Linguagem C, que é uma linguagem de programação com tipagem de dados forte, por isso a declaração correta dos nossos dados faz-se necessária.

Ao declararmos os tipos de dados, estamos informando ao computador como alocar e manipular tais dados em memória. A seguir serão apresentados alguns dos possíveis tipos para declaração.

Tipo de dados simples

Em Linguagem C os dados simples ou primitivos podem assumir os seguintes tipos, são:

- ***char:*** armazena um carácter, o que pode ser uma letra, número ou pontos;
- ***int:*** representa valores inteiros;
- ***float:*** utilizado para armazenar número em ponto flutuante de precisão simples, também são conhecidos normalmente como números reais;
- ***double:*** representa um número em ponto flutuante de precisão dupla;
- ***void:*** representa a ausência de tipo de dados. Nas linguagens de programação derivadas de C, que incluem C++ e Java, ***void*** é uma palavra-chave usada para identificar que uma função não retorna um resultado por meio de return. Ela é colocada no lugar de tipo de retorno da função. Em variações do C, void é usado, também, para indicar que uma função não recebe argumentos.

Além dos tipos simples, temos os modificadores, que são quatro:

- *signed;*
- *unsigned;*
- *long; e*
- *short.*

Tais modificadores são palavras que alteram o tamanho do conjunto de valores dos tipos de dados simples que estes podem representar. Por exemplo, um inteiro com o modificador *unsigned* é sempre um inteiro positivo. Ao declarar um dado com este tipo, a ele é permitido armazenar somente números inteiros maiores ou igual a zero. O Quadro 1 mostra os tipos de dados simples juntamente com os respectivos modificadores.

ATENÇÃO:
Quando uma variável é declarada com um tipo de dado, o valor atribuído a ela tem que ser exatamente do mesmo tipo declarado.

Quadro 1 – Tipos primitivos e seus modificadores

Tipo	Descrição	Tamanho
Bool	Valores aceitos: true ou false	1 byte ou 8 bits
Char	Intervalos de -128 a 127	1 byte ou 8 bits
signed char	Intervalos de -128 a 127	1 byte ou 8 bits
unsigned char	Intervalos de 0 a 255	1 byte ou 8 bits
Int	Intervalos de -2.147.483.648 a 2.147.483.647	4 bytes ou 32 bits
short int	Intervalos de -32.768 a 32.767	2 bytes ou 16 bits
signed int	Intervalos de -2.147.483.648 a 2.147.483.647	4 bytes ou 32 bits
unsigned int	Intervalos de 0 a 4.294.967.295	4 bytes ou 32 bits
unsigned short int	Intervalos de 0 a 4.294.967.295	2 bytes ou 32 bits
Float	Intervalos de 1,2e-38 até 3,4e+38.	4 bytes ou 32 bits
Double	Intervalos de 2,2e-308 a 1,8e+308	8 bytes ou 64 bits
Long	Intervalos de -2.147.483.648 a 2.147.483.647	4 bytes ou 32 bits
signed long	Intervalos de -2.147.483.648 a 2.147.483.647	4 bytes ou 32 bits
unsigned long	Intervalos de 0 a 4.294.967.295	4 bytes ou 32 bits

Fonte: Autores.

Tipo de dados compostos

A partir desses tipos de dados simples, podemos agrupá-los e formar outros tipos de dados, os quais denominamos de dados compostos. Sendo assim, os tipos de dados compostos, derivam dos tipos simples e são úteis para armazenar grandes quantidades de um tipo de dados, por exemplo, quando há a necessidade de armazenar uma frase inteira ou, os tempos dos atletas que disputaram uma corrida.

Nestes dois exemplos podemos criar tipos compostos que atendam às neces-

sidades de cada problema. Para armazenar uma frase inteira podemos usar o tipo char que armazena caracteres, como as letras 'a', 'b' e 'x', para formar um vetor de caracteres, que denominamos de string. Seguindo a mesma linha de raciocínio, podemos criar um vetor do tipo double para armazenar os tempos.

Veremos a seguir os dois principais tipos de dados compostos, são eles:

• Vetor: É um conjunto de dados do mesmo tipo. Os vetores podem ser do tipo numérico, que armazenam valores do tipo inteiro, tais como float, double, ou vetores que armazenam dados literais, que armazenam textos, podendo ser um caracter ou string.
• Matriz: Conceitualmente, uma matriz é o vetor de vetores ou um vetor com mais de uma dimensão.

Tipos Abstratos de Dados

Entendemos, por tipo abstrato de dados (TAD), a definição de coleções de dados a serem armazenados, juntamente com as suas respectivas operações.

Nesse momento, vamos abordar como podemos criar as coleções para armazenar os dados. Nesse sentido, ao implementar um TAD, na maioria das vezes, temos que escolher uma estrutura de dados para representá-lo computacionalmente. Cada tipo de estrutura de dados é construída por tipos simples (*inteiro, real* ou *char*) ou dos tipos compostos (*array* ou *struct*) de uma linguagem de programação.

Ao implementar um TAD podemos construir estruturas de dois tipos: Listas lineares e Listas não lineares:

• Lista linear: É uma série de elementos ordenados, na qual cada elemento, exceto o primeiro, possui um e apenas um antecessor, e cada elemento, exceto o último, possui um e apenas um sucessor, assim, existe apenas um caminho para percorrer a estrutura.
• Lista não linear: Preserva os conceitos básicos da lista linear, alterando a concepção das ligações entre os seus elementos, possibilitando a cada um deles ter mais de um sucessor. Assim, uma lista não linear é aquela em que existe mais de um caminho possível para percorrer a estrutura.

Seguindo o conceito de listas, a seguir são apresentadas as estruturas mais comuns, implementadas por meio de listas:

• Listas lineares
  - **Listas encadeadas:** É a representação de uma sequência de objetos alocados na memória do computador e, a cada novo objeto a ser inserido, é alocado dinamicamente um novo espaço na memória denominado nó ou nodo.
  - **Listas duplamente encadeadas:** É aquela em que cada nó possui duas autorreferências, uma para o nó anterior e a outra para o próximo nó.
  - **Filas:** É uma lista linear onde o primeiro elemento a entrar será o primeiro a sair.

- **Pilhas:** É uma lista linear onde todas as inserções, remoções e acessos são realizadas em um único extremo (topo da pilha).
- **Deques:** É uma lista linear onde as inserções, remoção ou acessos são realizados em qualquer um dos extremos.

• Listas não lineares:

- **Árvores:** É uma estrutura de dados em que cada elemento tem zero ou mais elementos associados.

Conforme visto na estrutura lista, apresentada no texto acima, ela está presente em todas as outras estruturas. Sendo assim, é muito importante que você aprenda bem a primeira estrutura apresentada, que é a lista encadeada, ela será a base para as demais.

Para se familiarizar mais com o assunto abordado, é aconselhável que você assista à vídeo-aula 1 e 2.

INTERATIVIDADE:
Assista às videoaulas sobre TAD:
Videoaula : TAD (Tipo Abstrato de Dado)
https://www.youtube.com watch?v=bryesHllovY&list=PL8iN-9FQ7_jt66JT04h_dcCduGW_WB4cPa&index=1

Videoaula: Modularização e TAD
https://www.youtube.com/watch?v=lKwEQgV6nZk&list=PL8iN9FQ7_jt66JT04h_dcCduGW_WB4cPa&index=2

1.2
ALOCAÇÃO DE MEMÓRIA

A alocação de memória consiste em um processo no qual o sistema operacional fornece a memória solicitada ao programa, para que ele seja executado ou carregado no computador (DEITEL et al., 2005). A Figura 1 apresenta a organização da memória no computador.

Figura 1 - Organização da Memória no computador

Fonte: Autores, 2018.

SAIBA MAIS:

Basicamente existem dois tipos de memória em um computador: Memória principal e memória secundária.

Quando alocamos memória estamos trabalhando com a memória principal.

Pesquise mais sobre a memória principal, por exemplo as memórias RAM e memórias Cache.

Na Figura 1 podemos perceber que existem três áreas destinadas para guardar os dados dos nossos programas, são elas: Dados, *heap* e pilha. Para que possamos entender como funcionam essas três áreas, podemos afirmar que toda vez que você executa um programa em seu computador, o código de máquina deste é carregado para a memória, além disso, o sistema operacional reserva também espaços em **Dados**, caso seja necessário. Neste espaço podem ser armazenadas as variáveis definidas como globais e estáticas.

Também, caso seja necessário, há a reserva de espaços no ***Heap***. Esta área se destina às variáveis alocadas de forma dinâmica. Além disso, o sistema operacional também reserva espaço na área denominada **Pilha**, ela é destinada às variáveis auto, ou também chamadas de variáveis de escopo local, ou simplesmente variáveis locais. Tanto o *Heap* quanto a Pilha compartilham o mesmo espaço na memória, desta

forma, o espaço que não for ocupado pela Pilha pode ser solicitado dinamicamente pelo seu programa na *Heap*, por meio de alocação dinâmica.

Como o espaço livre na memória é finito, e caso a pilha cresça além do espaço de memória disponível, dizemos que houve um **estouro** da pilha e o programa é finalizado imediatamente com um erro. Também pode ocorrer um erro similar com o *Heap*, nesse caso, o erro ocorre quando o espaço de memória livre for menor que o espaço requisitado dinamicamente, desta forma a alocação não é feita e o programa deve tratar o erro, informando ao usuário, por meio de mensagem, que a memória é insuficiente.

Neste sentido, dizemos que a alocação de memória no computador pode ser dividida em dois grupos principais:

• **Alocação Estática:** É aquela que reserva, ou aloca todo o espaço de memória a ser utilizado no início da execução do programa ou módulo, e não no decorrer da execução. Neste tipo de alocação, o espaço de memória permanece reservado durante toda a execução do programa, independentemente de estar sendo efetivamente utilizado ou não. Um exemplo típico desta alocação são as variáveis globais, *char c, int i* e *int v[10].*

• **Alocação Dinâmica:** Nesta alocação, o espaço de memória a ser utilizado, para armazenar os elementos, pode ser alocado no decorrer da execução de um programa ou módulo, quando for efetivamente necessário. Neste caso é importante ressaltar que os dados não precisam ter um tamanho fixo, pois podemos definir para cada um dos dados quanto de memória será utilizado. Desta forma vamos alocar blocos de memória que não precisam estar organizados de maneira sequencial, podendo estar distribuídos de forma dispersa na memória do computador. Na alocação dinâmica, basicamente vamos realizar 2 operações, alocação ou liberação de memória, de acordo com a nossa necessidade. Para poder gerenciar os blocos que estão dispersos ou espalhados na memória devemos usar as variáveis do tipo Ponteiro, que são indicadores de endereços de memória.

1.3
ALOCAÇÃO DINÂMICA

A alocação dinâmica é o processo que aloca memória em tempo de execução, permitindo a você, futuro programador, criar e alocar memória para novas variáveis quando o programa estiver em execução. Esta forma de alocação é utilizada quando não se sabe ao certo quanto de memória será necessário para o armazenamento das informações, podendo ser definida em tempo de execução, conforme a necessidade do programa, evitando, desta forma, o desperdício de memória. As funções de alocação dinâmica são muito eficientes para a implementação de estruturas de dados.

Em se tratando da linguagem C, que é a linguagem adotada neste livro, ela implementa com bastante eficiência o tratamento de alocação, demostrando o poder desta linguagem. Na linguagem C, no padrão ANSI, existem 4 funções para alocações dinâmica pertencentes à biblioteca *stdlib.h*. São elas *malloc(), calloc(), realloc() e free()*. Destas, as mais utilizadas são as funções *malloc() e free()*. A função *malloc()* é usada para solicitar a alocação de um bloco de memória. Esta função recebe a quantidade de bytes a serem alocados e retorna um ponteiro do tipo *void* (genérico) para o início do bloco de memória obtido. Para que possamos iniciar a compreensão dos conceitos da função *malloc*, vamos construir um exemplo simples, que solicite a alocação de memória para um dado do tipo inteiro. Sendo assim, o primeiro passo é definir um ponteiro para uma variável do tipo inteiro e, na sequência, solicitar a alocação de memória. Seguindo as premissas de alocação dinâmica, o código deve conter as linhas, abaixo:

```
int *ponteiro; //ponteiro para uma variável do tipo inteiro
ponteiro = malloc(4); //aloca memória para um int
```

Embora estas linhas de código estejam corretas, elas não representam boas práticas de programação. O problema está na função malloc pois ela assume que o tipo inteiro tem sempre 4 *bytes*. Para melhorar o código, vamos adotar o operador *sizeof()*, que retorna o tamanho de um tipo. Em C os tipos podem ser, por exemplo, *int, char, float,* etc. Neste exemplo, ela retorna à quantidade de *bytes* do tipo inteiro, conforme linhas abaixo:

```
int *ponteiro; // ponteiro para uma variável do tipo inteiro
ponteiro = malloc(sizeof(int)); // aloca memória para um int
```

Conforme já dito, a função *malloc()* retorna um ponteiro genérico para um tipo qualquer, representado por *void**, que pode ser convertido automaticamente pela linguagem para o tipo apropriado na atribuição. Para evitar problemas com diferentes compiladores, é comum fazermos a conversão explícita de dados para *malloc()*, utilizando o operador de conversão de tipo (*cast*), conforme segue:

ponteiro = ***(int *)*** *malloc(sizeof(int));//Conversão de tipo*

Agora que o nosso código ficou adequado, vamos passar a analisar a função *free()*. Ela é utilizada para liberar o espaço anteriormente alocado por *malloc()*, veja:

// libera a memória
free(ponteiro);

Nos próximos exemplos vamos adotar as funções *malloc()* e *free()* para trabalhar com alocação dinâmica de memória. A seguir vamos falar sobre as definições de ponteiros. Este conhecimento é muito importante para manipular os blocos de memória. Antes de prosseguir assista as videoaulas sobre alocação dinâmica.

INTERATIVIDADE:
Videoaula: Alocação Dinâmica - Parte 1 - Introdução
https://www.youtube.com/watch?v=ErOmueylikM

Videoaula: Alocação Dinâmica - Parte 2 – Sizeof
https://www.youtube.com/watch?v=p2ihD9uDZs4

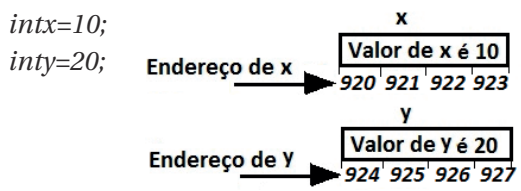
Videoaula: Alocação Dinâmica Parte 3 - MALLOC
https://www.youtube.com/watch?v=iU9CL5d-P5U

1.4
DEFINIÇÕES SOBRE PONTEIROS

Por definição, ponteiro é uma variável capaz de armazenar um número hexadecimal que corresponde a um endereço de memória de outra variável (SCHILDT, 1990). Normalmente, uma variável faz uma referência direta a um valor específico. Já um ponteiro contém um endereço de uma variável que armazena um valor específico. Sob esse ponto de vista, um nome de variável faz uma referência direta a um valor, e um ponteiro faz referência indireta a um valor.

Dizemos, então, que fazer referência a um valor por meio de um ponteiro é fazer uma referência indireta. Para cada tipo de dados existente em C, há um tipo de ponteiro capaz de armazenar o seu endereço. Conforme apresentado na Figura 2, temos duas variáveis do tipo inteiro alocadas na memória:

Figura 2 - Declaração das variáveis x e y



Fonte: Autores, 2018.

Conforme apresentado na Figura 2, quando declaramos uma variável inteira (*int x*), o computador reserva, geralmente, um espaço de 4 *bytes* na memória para que essa variável seja armazenada. Sendo assim, conforme exemplo da Figura 2, o conteúdo de x é armazenado no endereço 920, que é o endereço do primeiro *byte*, e o conteúdo de y é armazenado no endereço 924.

Agora vamos declarar P1 e P2 como 2 variáveis do tipo ponteiro para inteiros. Lembrando, para que elas sejam declaradas como ponteiros, temos que preceder o nome das variáveis com o caractere *, veja:

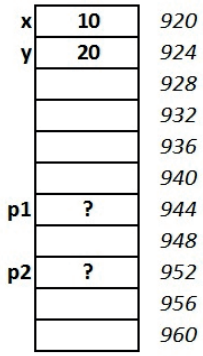
```
int *p1; //Ponteiro p1
int *p2; //Ponteiro p2
```

De acordo com a nossa declaração, estas variáveis, P1 e P2, estão aptas a armazenarem um endereço de memória em que exista um tipo inteiro armazenado. Logo, elas podem armazenar o endereço de x e de y, que foram declarados como sendo do tipo inteiro.

A Figura 3 mostra a pilha de execução para as variáveis *x, y, p1* e *p2*. A variável *x* foi armazenada no endereço 920, cujo valor armazenado é 10, e a variável y no endereço 924, com valor armazenado igual a 20. Já os ponteiros, ambos não possuem endereços de variável armazenada, logo não apontam para lugar algum, que é

representado por "?", pois ainda não foram inicializados. Para estes, foram alocados endereços do tipo inteiro, sendo atribuído a eles os endereços dos primeiros bytes alocados. Como ambos são do tipo inteiro, cada um ocupa 4 bytes. Portanto, o endereço 944 deve ser atribuído a p1 e 952 para p2.

Figura 3 - Pilha de execução

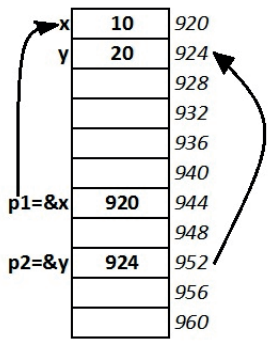


Fonte: Autores, 2018.

Para que possamos atribuir os endereços de *x* e *y* a *p1* e *p2*, temos que usar o operador unário &, que significa o endereço de. Após a aplicação do operador &, a pilha de memória apresenta as seguintes alterações, apresentadas na Figura 4.

Figura 4 - p1 e p2 recebem os endereços de x e y.

```
//p1 recebe o endereço de x
p1 = &x;

//p2 recebe o endereço de y
p2 = &y;
```



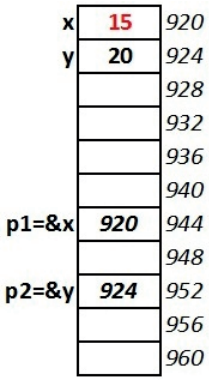
Fonte: Autores, 2018.

Conforme apresenta a Figura 4, o ponteiro p1 recebe o endereço de x. Também podemos afirmar que p1 aponta para x. O mesmo acontece com *p2*, ele recebe o endereço de *y*, logo o ponteiro p2 aponta para y.

Agora, vamos inserir um novo valor para *x*, por exemplo, 15, por meio do ponteiro *p1*. Para tal, teremos que usar o operador unário *, que nesta situação, significa conteúdo de. Ao aplicar este operador na variável p1, temos acesso ao conteúdo do endereço de memória armazenado por ela. Analisando a Figura 5, *p1* armazena o endereço 920, que é o endereço da variável x, cujo valor é 10. Logo, ao usarmos **p1*, teremos acesso ao conteúdo de *p1*, sendo este o endereço de *x*, que por vez, possui o valor 10.

Figura 5 - Atribuição de valor ao conteúdo de p1

//O conteúdo de p1 recebe o valor 15
**p1 =15;*



Fonte: Autores, 2018.

Conforme podemos observar na Figura 5, o conteúdo de *p1* é o endereço de x. Logo, ao executar *p1=15, estamos atribuindo 15 na posição memória 920, que é o endereço de x. Sendo assim, ao analisar as alterações dessa operação podemos concluir que o conteúdo da variável x, que antes era 10, agora é 15.

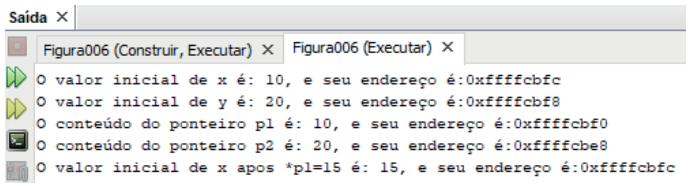
A Figura 6 apresenta o código em Linguagem C das ações descritas anteriormente. O código trata da atribuição, declaração, utilização do operador unário de endereço & (“recebe o endereço de”) e o operador unário de conteúdo * (“o conteúdo de”). O resultado da execução do programa da Figura 6 é apresentado na Figura 7.

Figura 6 - Código sobre ponteiros

```
#include <stdio.h>//Rotinas de entrada e saida (ler do teclado e imprimir no monitor)
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {
    int x=10;//declaração e atribuição da variável x
    int y=20;//declaração e atribuição da variável y
    int *p1;//declaração da variável p1 (ponteiro p/inteiro)
    int *p2;//declaração da variável p2 (ponteiro p/inteiro)

    p1=&x;//p1 recebe o endereço de x
    p2=&y;//p2 recebe o endereço de y

    printf("O valor inicial de x é: %d, e seu endereço é:%p \n",x,&x);
    printf("O valor inicial de y é: %d, e seu endereço é:%p \n",y,&y);

    printf("O conteúdo do ponteiro p1 é: %d, e seu endereço é:%p \n",*p1,&p1);
    printf("O conteúdo do ponteiro p2 é: %d, e seu endereço é:%p \n",*p2,&p2);

    *p1=15;
    printf("O valor inicial de x apos *p1=15 é: %d, e seu endereço é:%p \n",x,&x);
    return 0;
}
```

Fonte: Autores, 2018.

Figura 7 - Resultado da execução do código da Figura 6



Fonte: Autores, 2018.

Antes de prosseguir com a leitura, assista à videoaula, referente aos conceitos iniciais de ponteiro, para aprofundar seus estudos.

INTERATIVIDADE:
Assista a videoula sobre ponteiros:
Videoaula: Parte 1- Conceitos
https://www.youtube.com/watch?v=SJzd9x2S2yg

1.5
USO DE PONTEIROS

Qualquer programa escrito em Linguagem C, com a finalidade de resolver problemas práticos dificilmente dispensará o uso de ponteiros. Basicamente temos quatro possibilidades de usá-los, são elas:

- Manipulação de parâmetros de variáveis passados para as funções;
- Acesso às informações armazenadas em matrizes
- Manipulação de strings
- Alocação dinâmica de memória para criar estruturas dinâmicas de dados.

Muitos programadores tentam evitar o uso de ponteiros, por não gostar ou por ter dificuldades em trabalhar com eles, mas isso resulta, em muitas das vezes, em códigos maiores e de execução mais lenta. Por isso, vamos detalhar as principais formas de utilização.

Operações com ponteiros

Antes de utilizar os ponteiros é importante que você entenda a aritmética destes. Quando se utilizam ponteiros, não apenas o seu conteúdo pode ser usado em expressões aritméticas, mas também, há suporte ao conceito de operações sobre endereços, embora as operações que possam ser utilizadas, neste caso sejam limitadas, pois temos a adição e subtração.

Os ponteiros podem aparecer em expressões de adição, por exemplo:

```
int x=10;
int *p1;
p1 = &x;
y=*p1+1;//Incrementa 1 no conteúdo de p1 igual a y=x+1
```

Se **p1** aponta para um inteiro x, então ***p1** pode ser utilizado em qualquer lugar que x seria usado. O operador * tem maior precedência que às operações aritméticas. Assim a expressão acima, pega o conteúdo do endereço que **p1** aponta e soma 1.

No próximo exemplo, o endereço do ponteiro **p1** é incrementado uma unidade e o conteúdo do novo endereço é atribuído à variável y:

```
y=*(p1+1); //Incrementa uma posição na memória
```

Os incrementos e decrementos dos endereços podem ser realizados com os operadores ++ e --, que possuem procedência sobre o * e operações matemáticas, sendo avaliados da direita para a esquerda:

p1++;//Incrementa uma posição na memória.
*(p1--);//Decrementa uma posição, mesma coisa de *p1--*

No exemplo seguinte, veja que foram inseridos parênteses antes da operação de soma. Estes são necessários quando queremos incrementar o conteúdo de p1, sem os parênteses, o que seria incrementado é a posição de memória, pois os operadores * e ++ são avaliados da direita para esquerda.

*(*p1)++ // equivale a x=x+1; ou *p1+=1*

As operações entre ponteiros podem gerar muitos erros, de difícil solução. Para entender melhor essas operações assista à videoaula referente a estes conceitos.

INTERATIVIDADE:
Assista à videoaula sobre operações de ponteiros:
Videoaula: Parte 2 - Operações
https://www.youtube.com/watch?v=cg1mnWupbTE&t=18s

Erros Comuns ao utilizar ponteiros

Um dos problemas que levaram à extinção do uso de ponteiros em linguagens de programação mais recentes, tais como o Java, que são baseadas em C, são os erros de lógica, que muitas vezes levam o programa ao crash ou erro fatal. Um dos erros mais comuns em ponteiros é a falta da inicialização. Ponteiros devem sempre apontar para algum endereço de memória. Portanto, todo ponteiro deve ser inicializado antes de ser utilizado. Veja o código de exemplo na Figura 8.

Figura 8- Erros comuns: falta de inicialização do ponteiro p1

```
14  #include <stdio.h>
15  #include <cstdlib>
16
17  using namespace std;
18
19  /*...3 linhas */
22  int main(int argc, char** argv) {
23      int x;
24      int *p1; /*um ponteiro para um inteiro*/
25      //p1=&x; //p1 não é inicializado com o endereço de i
26      *p1=12;
27      printf("O conteúdo de x: %d, o conteúdo de p é: %d",x,*p1);
28
29      return 0;
30  }
```

Fonte: Autores, 2018.

Conforme mostra a Figura 8, o ponteiro **p1** foi declarado como sendo do tipo inteiro, e logo na linha 26 (vinte e seis), o conteúdo de p1, recebe o valor 12. Mas o ponteiro não foi inicializado. Desta forma, ele aponta para um local desconhecido na memória, podendo ser para a pilha do sistema, para variáveis globais ou até mesmo para um espaço de código do programa.

Desta forma, ao atribuir ***p=12**, o programa tenta atribuir o número 12 a qualquer local para onde **p** apontar. Como **p** não aponta para um lugar conhecido, o programa vai dar um erro imediatamente, ou após algum tempo, ou ainda, pode corromper sutilmente os dados em qualquer outra parte do programa, sem que você nunca perceba. Por não saber a consequência exata da falta de inicialização do ponteiro, a correção deste erro torna-se bastante difícil de ser rastreada. Sendo assim, é importante frisar que ponteiros não guardam valores, apenas endereços de memória. Portanto, você deve se certificar que todos os ponteiros foram inicializados com um endereço válido. Caso não tenha inicializado, o faça, pois o seu programa pode apresentar erros gravíssimos e difíceis de localizar.

Utilização de ponteiros como argumento para funções

Ao passarmos argumentos para funções temos dois tipos de passagem: a passagem por valor e passagem por referência. Na **passagem por valor**, é repassada à função uma cópia do valor da variável como parâmetro no momento da chamada, impedindo que a função manipule diretamente a mesma, protegendo desta forma o seu conteúdo, ou seja, apenas a função onde as variáveis foram declaradas pode modificar seu conteúdo.

Já na **passagem por referência**, o endereço da variável é passado na chamada da função, permitindo que ela modifique a variável diretamente. O uso da passagem por referência é aconselhável quando temos funções que devem retornar mais de um valor.

Para entender como os parâmetros por referência funcionam, veja o programa apresentado na Figura 9, que implementa a função **troca** em C. Esta função recebe dois argumentos do tipo inteiro e troca seus valores.

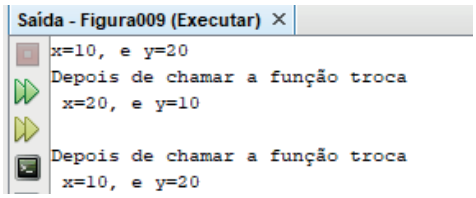
Figura 9- Função troca

```
13  #include<stdio.h>
14  #include <cstdlib>
15
16  using namespace std;
17
18  //...5 linhas
23  void troca(int *px, int *py){
24      int n;
25      n=*py;
26      *py=*px;
27      *px=n;
28  }
29
30  int main(int argc, char** argv) {
31      int x=10;
32      int y=20;
33
34      printf("x=%d, e y=%d \n",x,y);
35
36      //passagem do endereço das variáveis x e y
37      troca(&x,&y);
38      printf("Depois de chamar a função troca\n x=%d, e y=%d\n\n",x,y);
39      troca(&x,&y);
40      printf("Depois de chamar a função troca\n x=%d, e y=%d\n\n",x,y);
41      return 0;
42  }
```

Fonte: Autores, 2018.

Conforme exposto na Figura 9, é preciso inserir o caracter **&** durante a passagem dos argumentos para a função **Troca**, na linha 19. Este caracter indica que a passagem do argumento é por referência, logo ela passa o endereço de x e y, proporcionando a alteração dos seus conteúdos pela função chamada. Na linha 4, que é a assinatura da função Troca, os parâmetros devem ser precedidos pelo caractere *, que significa que eles estão recebendo o endereço das variáveis. O resultado da execução do código da Figura 9 é apresentado na Figura 10.

Figura 7 - Resultado da execução do código da Figura 6



Fonte: Autores, 2018.

Utilização de ponteiros com Vetores e Matrizes

É importante ressaltar que vetores (ou matrizes unidimensionais), matrizes bidimensionais e matrizes de qualquer dimensão são caracterizados por terem todos os elementos pertencentes ao mesmo tipo de dado. Vejamos, a seguir, a declaração de uma matriz:

int nomeVetor[10];

Quando o compilador C executa esta linha ele calcula o tamanho, em *bytes,* necessário para armazenar este vetor. O cálculo é feito multiplicando o total de elementos a serem armazenados no vetor pelo tamanho do tipo inteiro. Sendo assim o cálculo do total de *bytes* necessários para armazenar é:

tam1 x tamanho_do_tipo = 10 x 4

Ao final, o compilador aloca 40 *bytes* em um espaço livre de memória. O nome da variável do vetor é na verdade um ponteiro para o tipo da variável *nomeVetor* que aponta para o primeiro elemento. Este conceito é fundamental. Por meio dele é possível justificar por que o nome da variável, que é um ponteiro, aponta para o primeiro elemento da matriz. Mas ainda há um questionamento, como é que se pode usar a seguinte notação?

nomeVetor [i];

Isto pode ser facilmente explicado desde que você entenda que a notação acima é absolutamente equivalente à:

**(nomeVetor +índice)*

Agora é possível entender como funciona um vetor. Fica claro, por exemplo, porque é que, na linguagem de programação C, a indexação começa com zero pois, ao usar o valor do primeiro elemento de um vetor, temos, somente o **nomeDaVariável,* logo o índice deve ser igual a zero. Veja:

** nomeVetor é equivalente a nomeVetor [0]*

Para exemplificar o uso de ponteiros com vetores, a Figura 11 ilustra a passagem do vetor como argumentos para a função soma.

Com o objetivo de exemplificar ainda mais, assista à videoaula, , antes de iniciar a análise do código, da Figura 11.

INTERATIVIDADE:
Assista a videoaula: Função Parte 7 - Array como parâmetro
https://www.youtube.com/watch?v=SAhR1h3LpDY

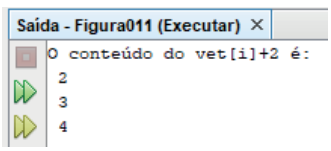
Figura 11- Passagem de vetor para função

```
14  #include <cstdlib>
15  #include <stdio.h>
16
17  using namespace std;
18
19  /*...3 linhas */
22  void soma(int *v){
23      int i;
24      for(i=0;i<3;i++){
25          *(v+i)+=2;//equivalente a v[i]=v[i]+2
26      }
27  }
28
29  int main(int argc, char** argv) {
30      int vet[]={0,1,2};
31      soma(vet);
32      printf("O conteúdo do vet[i]+2 é:\n %d\n %d\n %d\n",vet[0],vet[1],vet[2]);
33      return 0;
34  }
```

Fonte: Autores, 2018.

Na linha 31 (Figura 11) é realizada a chamada para a função soma, passando como parâmetro o nome do vetor declarado como *int vet*. Ao analisarmos a assinatura da função soma, na linha 22, o parâmetro passado é recebido como ponteiro e na linha 25 é somado e atribuído 2 a cada elemento do vetor, no índice *i*. A Figura 12 apresenta o resultado da execução do referido programa.

Figura 12- Resultado da execução do programa da Figura 11.



Fonte: Autores, 2018.

Veja, na Figura 12, que os valores do vetor vet estão diferentes quando comparamos com os valores inseridos na linha 30 da Figura 11 pois, ao passar o vetor para a função ocorre a passagem por referência do primeiro elemento, o que possibilita a alteração dos valores por parte da função chamada soma.

Ao declarar o vetor da Figura 11, obrigatoriamente você tem que informar a sua dimensão, sendo necessário prever o número máximo de elementos que o vetor irá armazenar durante a codificação para todas as aplicações em que o programa for utilizado. Em algumas aplicações o número máximo pode faltar e em outra exceder, para resolver esse problema vamos usar alocação dinâmica para vetores. A Figura 13 apresenta a alocação dinâmica de uma matriz unidimensional (vetor), usando a função *malloc()*.

Figura 13- Alocação dinâmica de matriz unidimensional (vetor)

```
14  #include <cstdlib>
15  #include <stdio.h>
16  #include <stdlib.h>
17
18  using namespace std;
19
20  /*...3 linhas */
23
24  void soma(int totalElementos, int *matriz){
25      int i;
26      for(i=0;i<totalElementos;i++){
27          *(matriz+i)+=2;//Equivalente a matriz[i]=matriz[i]+2
28      }
29  }
30
31  int main(int argc, char** argv) {
32      int *ponteiroParaMatriz;
33      int linhas, colunas;
34      int i;
35      printf("\nDigite o numero de linhas da matriz: ");
36      scanf("%d",&linhas);
37      printf("\nDigite o numero de colunas da matriz: ");
38      scanf("%d",&colunas);
39
40      //alocaçao dinâmica da matriz[linhas][colunas]
41      ponteiroParaMatriz=(int*)malloc(linhas*colunas*sizeof(int));
42      if(ponteiroParaMatriz==NULL){
43          printf("Memoria insuficiente.\n");
44          return 1;
45      }
46
47      /*Leitura dos elementos da matriz*/
48      for(i=0;i<linhas*colunas;i++){
49          printf("\n Digite o %d elento da matriz:",i);
50          scanf("%d",&ponteiroParaMatriz[i]);
51      }
52
53      soma(linhas*colunas,ponteiroParaMatriz);
54
55      //imprime os elementos da matriz somados a 2
56      for(i=0;i<linhas*colunas;i++){
57          printf("\n O conteudo da Matriz[i]+2 e: %d",ponteiroParaMatriz[i]);
58      }
59
60      free(ponteiroParaMatriz);
61      return 0;
62  }
```

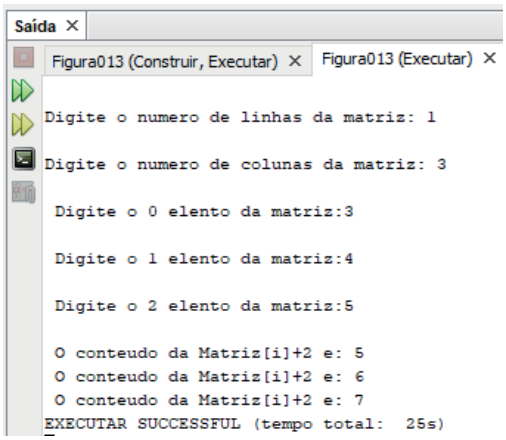
Fonte: Autores, 2018.

Quando executamos um programa, e o compilador C encontra a declaração de uma matriz, por exemplo, a matriz[linhas][colunas] do tipo inteiro, ele calcula o tamanho, em *bytes*, que é necessário para armazená-la da seguinte forma:

linhas x colunas x tamanho do tipo inteiro

Por isso, na linha 41 da Figura 13, a função *malloc* foi definida como sendo: *malloc(linhas*colunas*sizeof(int))*. Os valores das linhas e colunas são fornecidos pelo usuário em tempo de execução. O resultado da execução do programa apresentado na Figura 13 é mostrado na Figura 14.

Figura 14- Passagem de vetor para função



Fonte: Autores, 2018.

Conforme mostra a Figura 14, o usuário definiu uma *matriz[1][3]*. Desta forma foi definida uma matriz unidimensional ou simplesmente vetor, e foram inseridos os elementos 3, 4, e 5. A diferença do exemplo da Figura 11 é que lá a alocação do número de elementos é fixa. Portanto, o programa só trabalha com a dimensão do vetor declarada, enquanto que, no exemplo da Figura 13 a alocação do número de elementos é dinâmica, o que possibilita em tempo de execução definir o tamanho exato do vetor ou matriz.

Utilização de ponteiros com String

Uma string é considerada como sendo um vetor de caracteres terminado com um caractere nulo que, em linguagem C, por convenção, é representado por '\0'. Para declarar uma *string*, podemos usar um vetor de caracteres ou ponteiros, conforme é mostrado nos fragmentos de código da Figura 15.

Figura 15 - Ponteiro para string

```
10      //Ponteiro para strings
11      char *ponteiroStr;
12      //atribuição da string teste ao ponteiroStr
13      ponteiroStr=(char*)"teste de string com ponteiro";
14      imprimeString(ponteiroStr);
```

Fonte: Autores, 2018.

Levando em consideração o fragmento de código da Figura 15, quando ele for compilado, o compilador cria o arquivo objeto que contém seu código de máquina e uma tabela com todas as constantes do tipo strings declaradas no programa, sendo esta inserida em *Dados*, *Heap* ou *Pilha*, de acordo com o tipo (global, estática, dinâmica ou local) de declaração da variável *string*.

Na linha 11(Figura 15) é declarado um ponteiro do tipo string. Logo abaixo, na linha 13, tem-se a instrução ***ponteiroStr=(char*)"teste de string com ponteiro"***; ela faz com que o *ponteiroStr* aponte para o endereço da *string* de caracteres "***teste de string com ponteiro***" na tabela de constantes de *string*. Como esta *string* está na tabela de constantes de *string*, tecnicamente parte do código executável, não pode ser modificado. Você só pode apontar para ela e utilizá-la para leitura. Portanto, o código da Figura 15 funciona perfeitamente.

De acordo com a figura 16, foi declarado um vetor de caracteres *char vetString[100]*; conforme linha 17. A função gets, na linha 19, copia o conteúdo digitado para o *vetor vetString*. Caso a linha 19 fosse substituída ***por vetString=(char*)"teste de string com ponteiro"***; o programa não funcionaria, pois vetString não é um ponteiro e a "***teste de string com ponteiro***" está inserida na tabela. Portanto, deve ser repassado o seu endereço, o que exige um ponteiro para recebê-lo. Desta forma o código apresentado na Figura 16 funciona corretamente.

Figura 16- String como vetor de caracteres

```
16      //String como vetores de caracteres
17      char vetString[100];
18      printf ("Digite uma string: ");
19      gets (vetString);
20      imprimeString(vetString);
```

Fonte: Autores, 2018.

O código apresentado na Figura 17 compila perfeitamente, mas ao executá-lo ocorre um erro, ocasionando a interrupção imediatamente do programa. O erro ocorre na linha 28, ao executar a chamada da função para liberar o espaço de memória alocado, que é a ***free(ponteiroStrMalloc)***. Quando o programa executa a linha 24, a função *malloc* aloca um bloco de memória de 100 *bytes* para o tipo char e atribui o endereço a *ponteiroStrMalloc*. Porém, ao executar a linha 26, ***ponteiroStrMalloc=(char*)"teste de string com ponteiro"***, embora a sua sintaxe esteja correta, esse fragmento de código atribui a *ponteiroStrMalloc* o endereço da string "***teste de string com ponteiro***", deixando sem nenhum apontamento o bloco de memória alocado na linha 24. Sendo assim, é dito que o bloco de memória ficou órfão. Considerando que *ponteiroStrMalloc* está apontando para a tabela de constante de *string*, a *string* de caracteres não pode ser alterada em ***free(ponteiroStrMalloc)*** e ocorre a falha, pois não é possível desalocar um bloco na região executável. Para que o código apresentado na Figura 17 funcione temos que usar a função *strcpy*, comentada na linha 25.

Figura 17 - Alocação dinâmica de string com ponteiros

```
22      //Alocação dinamica de string com ponteiros
23      char *ponteiroStrMalloc;
24      ponteiroStrMalloc=(char *) malloc(100*sizeof(char));
25      //strcpy(ponteiroStrMalloc,"Teste alocacao dinamica");
26      ponteiroStrMalloc=(char*)"teste de string com ponteiro";
27      imprimeString(ponteiroStrMalloc);
28      free(ponteiroStrMalloc);
```

Fonte: Autores, 2018.

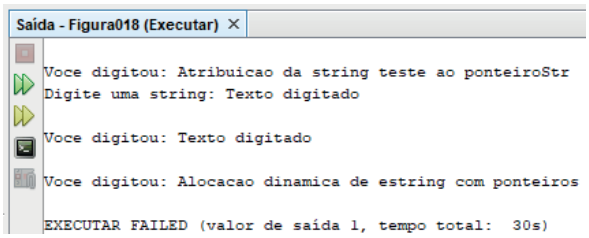
A Figura 18 apresenta os códigos, comentados anteriormente, com as devidas correções.

```
14  #include <cstdlib>
15  #include <stdio.h>//para printf()
16  #include <stdlib.h>//para malloc()
17  #include <string.h>//para strcpy()
18
19  using namespace std;
20
21
22  /*...3 linhas */
25
26  void imprimeString(char *str){
27      printf("\nVoce digitou: %s \n",str);
28  }
29
30  int main(int argc, char** argv) {
31      //Ponteiro para string
32      char *ponteiroStr;
33
34      //Atribuição da string teste ao ponteiroStr
35      ponteiroStr=(char*)"Atribuicao da string teste ao ponteiroStr";
36      imprimeString(ponteiroStr);
37
38      //String como vetores de caracteres
39      char vetString[100];
40      printf("Digite uma string: ");
        gets(vetString);
42      imprimeString(vetString);
43
44      //Alocação dinâmica de estring com ponteiros
45      char *ponteiroStrMalloc;
46      ponteiroStrMalloc=(char*)malloc(100*sizeof(char));
47      //strcpy(ponteiroStrMalloc,"teste aloção dinâmica");
48      ponteiroStrMalloc=(char*)"Alocacao dinamica de estring com ponteiros";
49      imprimeString(ponteiroStrMalloc);
50      free(ponteiroStrMalloc);
51
52      return 0;
53  }
```

Fonte: Autores, 2018.

A Figura 19 apresenta o resultado da execução do programa mostrado na Figura 18.

Figura 19- Resultado da execução do programa da Figura 18



Fonte: Autores, 2018.

Atividades – Unidade 1

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) De acordo com a Figura 20, faça o teste de mesa e descubra qual o valor final de x.

Figura 20 - Código sobre aritmética de ponteiros.

```c
#include <stdio.h>

int main(void){
    int x,*px,*py;
    x=9;
    px=&x;
    py=px;

    printf("O conteudo de  x= %d\n",x);
    printf("O endereco de &x= %d\n",&x);
    printf("O endereco de px= %d\n",px);
    printf("O endereco de py= %d\n",py);
    py++;//Incremento no endereço de py

    printf("O incremento no endereco de py= %d\n",py);

    (*px)++;//Incrementa o conteudo de px
    printf("Conteudo de *px= %d\n",*px);
    printf("Conteudo de *py= %d\n",*py);
}
```

Fonte: Autores, 2018.

2) Escreva um programa, em linguagem C, que converta uma sequência de caracteres maiúsculos para minúsculo, utilize alocação dinâmica com a função malloc. Por exemplo: FREDERICO WESTPHALEN deve ser convertida para frederico westphalen.

2

LISTAS LINEARES

INTRODUÇÃO

As listas se apresentam como um mecanismo versátil para serem utilizados em muitos tipos de bases ou estruturas de dados. Por ter essa versatilidade, ela se tornou uma das formas mais comuns de agrupar dados. Podemos entender que uma lista linear é uma estrutura de dados na qual os elementos, de um mesmo tipo de dado, estão organizados de forma sequencial (CELES, 2004). Neste panorama, é importante ressaltar que uma lista linear permite representar um conjunto de dados afins, sendo estes de um mesmo tipo, de forma a preservar a relação de ordem entre seus elementos. Cada elemento da lista é chamado de nó, ou nodo.

Conceitualmente, uma lista linear é composta por um conjunto de N nós, onde $N \geq 0$, e $x_1, x_2, \ldots, x_n$, são organizados de forma a preservar a relação de ordem entre cada um dos nós. Se $N \geq 0$, então x_1 é o primeiro nó. Para $1 < k < n$, o nó x_k é precedido pelo nó x_{k-1} e seguido pelo nó x_{k+1} e, xn é o último nó. Quando $N = 0$, diz-se que a lista está vazia.

Quanto à forma de alocar memória para armazenar seus elementos, uma lista linear pode ser classificada em:

• **Sequencial ou Contígua:** Neste tipo de lista, os nós, além de estarem em uma sequência lógica, estão também fisicamente em sequência, sendo implementada usando vetor, ou também chamado de alocação estática.

• **Encadeada:** Os elementos não estão necessariamente armazenados sequencialmente na memória, porém a ordem lógica entre os elementos que compõem a lista deve ser mantida, sendo implementada, na maioria das vezes, usando locação dinâmica de memória.

Existem diversas operações que podem ser realizadas com listas lineares, as mais comuns são:

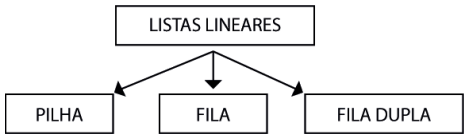
• Criação de uma lista;
• Remoção de uma lista;
• Inserção de um elemento na lista;
• Remoção de um elemento na lista;
• Acesso de um elemento na lista.

Rotineiramente, podemos criar programas usando esta estrutura para resolver diversos problemas práticos, tais como:

• Lista Telefônica;
• Lista de clientes de uma agência bancária;
• Lista de notas dos alunos de uma turma;
• Lista de itens em estoque em uma empresa.

Ao modelar aplicações computacionais usando lista, tem-se alguns casos especiais. O que os torna especiais são as operações de acesso, inserção e remoção. Contudo, eles recebem também nomes especiais, conforme mostra a Figura 21.

Figura 21- Casos especiais de lista linear



Fonte: Autores, 2018.

Dada as características de cada um destes casos, que serão detalhadas na próxima unidade, a diferença básica entre eles é apresentada abaixo:

• **Pilhas:** É uma lista linear onde todas as inserções, remoções e acessos são realizadas em um único extremo (topo da pilha). Uma pilha é denominada *LIFO - Last In First Out*, traduzindo, o último elemento que entrou, é o primeiro a sair.
• **Fila:** É uma lista linear do tipo *FIFO - First In First Out*, traduzindo, o primeiro elemento a entrar será o primeiro a sair. Na fila os elementos entram por um lado e saem por outro;
• **Fila dupla:** É uma lista linear onde as inserções, remoção ou acessos são realizados em qualquer extremo. Filas duplas são chamadas, também, de DEQUE (*Double-Ended QUEue*)

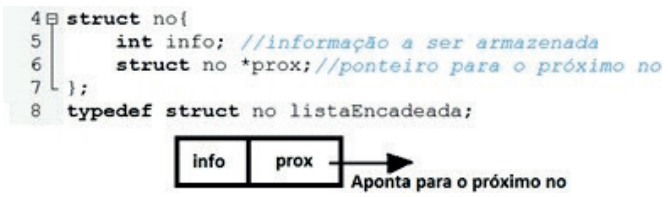
2.1
LISTAS ENCADEADAS

Por definição, uma lista encadeada é a representação de uma sequência de objetos alocados na memória do computador e, a cada novo objeto a ser inserido é alocado dinamicamente um novo espaço na memória (ZIVIANI, 2007), (UNICAMP, 2013).

Como iremos usar o mecanismo de alocação dinâmica, as listas encadeadas ocuparão sempre um espaço de memória variável, dependendo exclusivamente da quantidade de elementos a serem armazenados. Este tipo de lista é uma coleção de nós, que armazenam dados, e ligações para os outros nós. Os nós podem estar localizados em qualquer lugar na memória, e a passagem de um nó para o outro se faz por meio das ligações ou endereços de memória. Sendo assim, cada nó tem que conter um ou mais campos responsáveis pelo armazenamento das informações, além de um campo que é um ponteiro para armazenar uma ligação para uma próxima estrutura de um nó.

A Figura 22 apresenta a estrutura de lista autorreferenciada, com dois campos: um inteiro e, o outro um ponteiro *prox*, que aponta para a próxima estrutura do nó.

Figura 22- Estrutura e representação gráfica de um nó na lista encadeada



Fonte: Autores, 2018.

As listas encadeadas são largamente utilizadas em diversas aplicações. Basicamente, são utilizadas quando há necessidade de se criar vetores com tamanho variável, ou dinâmico. O seu sucesso se deve à simplicidade de implementação, além de fornecer bons resultados na gerência de informações, na alocação de memória e em trabalhos com conjuntos de dados não muito extensos.

A limitação, desta estrutura, está na forma de percorrê-la, pois ela permite somente o caminhamento unidirecional. Com isso, as referências iniciam no primeiro elemento da lista e seguem unidirecionalmente até o último elemento. Logo não é possível percorrer o caminho de volta através da lista. Ao construir uma lista encadeada, por meio dos nós, temos que realizar algumas operações, são apresentadas as principais:

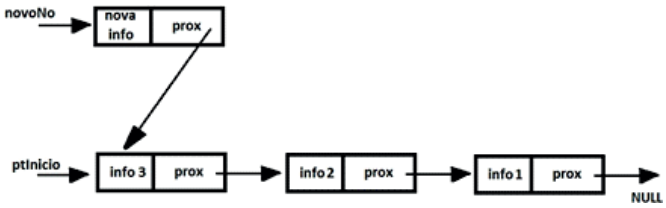
- Inserção: insere um novo dado no início da lista;
- Impressão: visita todos os nós da lista imprimindo o seu campo info (seu conteúdo);
- Busca: realiza uma busca sequencial em toda a lista, verificando a existência de um determinado elemento na mesma;
- Exclusão: remove um determinado elemento da lista.

A seguir serão detalhadas essas operações, por meio de programas de exemplo e representações gráficas.

2.1.1 Inserção no início da lista

A partir do momento em que a lista criada é inicializada, temos que proceder com a inserção de novos elementos. Lembrando que, para cada novo elemento adicionado na lista, é alocado um novo espaço de memória de forma dinâmica. A função de inserção apresenta algumas variações, de acordo com o local onde o novo elemento será inserido, por exemplo, ele pode ser inserido no início ou em outras localizações. Entretanto, a maneira mais simples é realizar a inserção sempre no início da lista. Desta forma o novo elemento entrará logo após o elemento marcado como o início da lista. A Figura 23 mostra, graficamente, como fica a inserção.

Figura 23-Inserção no início da lista



Fonte: Autores, 2018.

As linhas de código para executar a sequência de ações esquematizadas graficamente na Figura 23 estão apresentadas na Figura 24.

Figura 23-Inserção no início da lista

```
29  /*
30   * Insere um no início da lista e um ponteiro que armazena o endereço do inicio
31   * da lista
32   * iLista: ponteiro para o início da lista
33   * info: informaç]ao a ser armazenada no novo nó
34   */
35  listaEncadeada* insere(listaEncadeada* iLista, int info){
36      listaEncadeada* novoNo = (listaEncadeada*) malloc(sizeof(listaEncadeada));
37      novoNo->info = info;
38      novoNo->prox =iLista;
39      return novoNo;
40  }
```

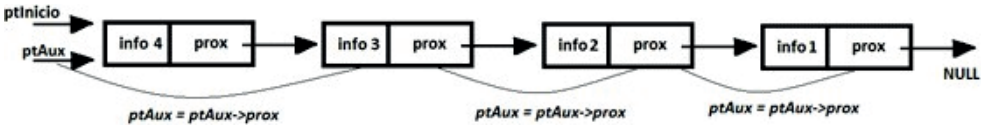
Fonte: Autores, 2018.

Inicialmente, temos que alocar espaço na memória para o novo nó. Essa tarefa é executada na linha 36 (Figura 24), após a sua criação, na linha 37 (novoNo->info = info), faz a inserção da informação que se deseja armazenar. Por fim é realizada a atualização de endereços do novo nó, na linha 38 (novoNo->prox = iLista). Desta forma, o novoNo->prox passa a apontar para o endereço de iLista, que é um ponteiro para o início da lista. Finalmente retorna-se o endereço de memória do novoNo, na linha 39, que agora aponta para o primeiro elemento da nossa lista, que deve ser atualizado.

2.1.2 Impressão da lista

A função de impressão possui uma tarefa bem simples, seu papel é percorrer toda a lista a partir do primeiro elemento, até chegar ao final: isto ocorre quando o ponteiro que percorre a lista encontra o valor NULL. Graficamente a impressão é representada pela Figura 25.

Figura 25- Impressão da lista



Fonte: Autores, 2018.

A Figura 26 mostra o código para executar a impressão dos elementos. Inicialmente, para que possamos percorrer toda a lista é criado um ponteiro auxiliar ptAux, com o objetivo de preservar o endereço do ponteiro do primeiro elemento da lista, o iLista, pois não podemos correr o risco de perder esse endereço. Antes de iniciar a impressão propriamente dita, é preciso apontar o ptAux para o primeiro endereço da lista, e verificar se ela não está vazia, além de definir a condição de parada e incremento, conforme mostra a linha 48. A impressão propriamente dita das informações ocorre na linha 49.

Figura 26-Código para a função de impressão

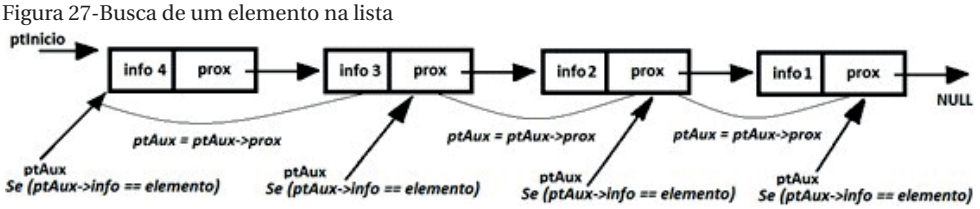
```
42  /*
43   * Imprime o campo info dos nós que foram armazenados na lista
44   * iLista: ponteiro para o início da lista
45   */
46  void imprime (listaEncadeada* iLista){
47      listaEncadeada* ptAux; // ponteiro auxiliar para percorrer a lista
48      for(ptAux = iLista; ptAux != NULL; ptAux=ptAux->prox){
49          printf("info = %d \n",ptAux->info);//para funcionar tem q incluir stdio.h
50      }
51  }
```

Fonte: Autores, 2018.

2.1.3 Busca um elemento na lista

Para verificarmos se um determinado elemento encontra-se presente na lista, temos que realizar uma busca sequencial na mesma. Em nosso exemplo, é verificado se um elemento pertence à lista. Desta forma, verificamos se o elemento é igual ao conteúdo de todos os nós na lista. Graficamente a busca é representada na Figura 27.

Figura 27-Busca de um elemento na lista



Fonte: Autores, 2018.

A função busca se assemelha à função imprime, a diferença básica entre elas é o teste de verificação de existência do elemento buscado em cada um dos nós.
O código para fazer a busca sequência na lista é apresentado na Figura 28. Veja que ele é bem semelhante ao da Figura 26. A função busca recebe, como parâmetro, um ponteiro iLista (que por sua vez aponta para o início da lista) e o elemento do tipo inteiro a ser buscado. Na linha 59 é criado um ponteiro denominado ptAux com a finalidade de percorrer a lista e verificar se o conteúdo dos nós é igual ao elemento procurado. Na linha 60 o ptAux recebe o endereço e iLista, o que faz com que ele aponte para o início da lista, e também recebe as condições de parada e incremento. Na linha 61 é realizada a verificação do elemento procurado com cada um dos nós, para saber se eles são iguais. Caso seja, então o elemento pertence à lista e, na linha 48, a função retorna o endereço no respectivo nó, que está armazenado em ptAux. Se o elemento não pertence à lista, na linha 49, a função retorna NULL.

Figura 28- Código para a função de busca

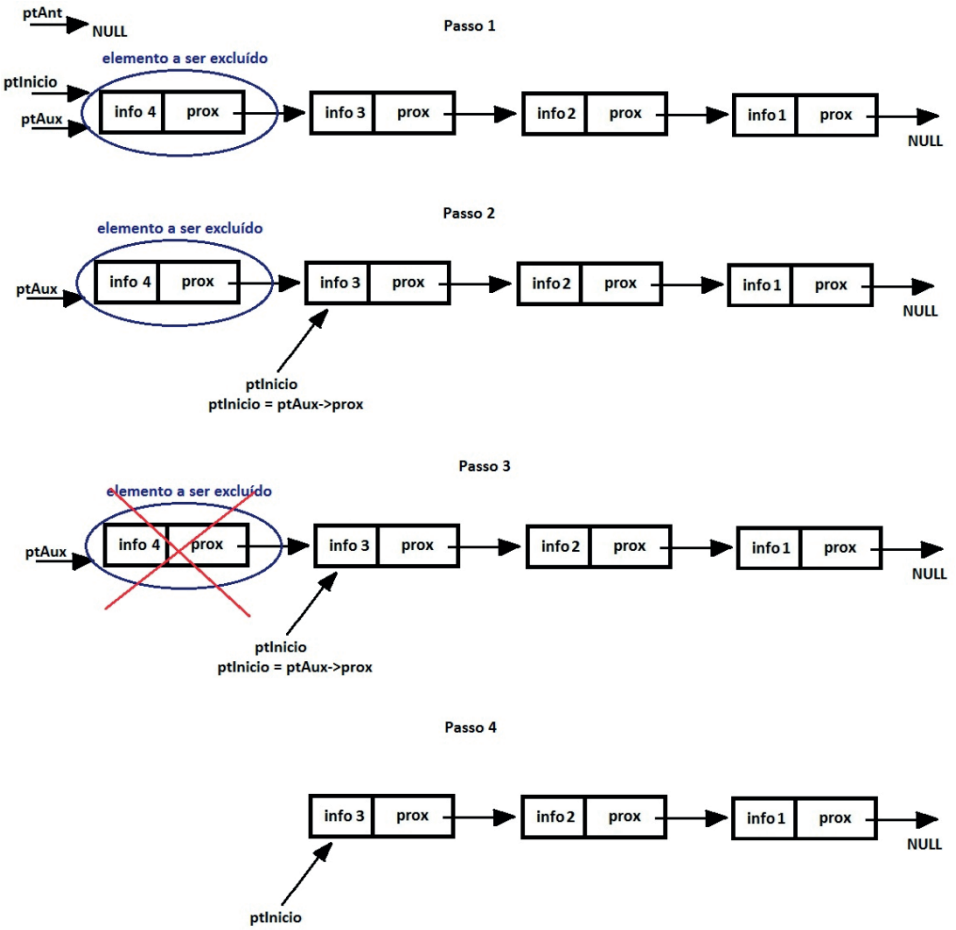
```
53  /*Busca um elemento na lista e retorna o endereço do nó que
54   * contém o elemento. Se tal nó não existe, a função devolve NULL
55   *  iLista: ponteiro para o início da lista
56   *  elemento: informação a ser buscada na lista
57   */
58  listaEncadeada* busca(listaEncadeada* iLista, int elemento){
59      listaEncadeada* ptAux;
60      for(ptAux = iLista; ptAux != NULL; ptAux=ptAux->prox){
61          if(ptAux->info == elemento){
62              return ptAux;
63          }
64      }
65      return NULL;//Não encntrou o elemento
66  }
```

Fonte: Autores, 2018.

2.1.4 Exclusão de um elemento da lista

Esta função é responsável por localizar e excluir um determinado elemento da lista. Nesta função é importante estar atento a alguns detalhes da lista, por exemplo, se ela está vazia, se o elemento a ser excluído é o primeiro ou os demais. A função para remover um elemento da lista é um pouco mais complexa, quando comparada com as anteriores, pois nela é preciso guardar o endereço do elemento anterior ao procurado, quando o elemento a ser excluído não for o primeiro, e fazer com que este aponte, após a exclusão do elemento identificado, para o posterior a ele, de forma a manter de maneira correta o encadeamento da lista. Graficamente a operação de remoção do elemento que se encontra no primeiro nó é representada na Figura 29. A operação pode ser compreendida nos seguintes passos: 1-localização do elemento a ser excluído, 2-apontar o ptInicio para o segundo elemento, 3-excluir o elemento e 4-atualizar a lista.

Figura 29- Exclusão do primeiro elemento da lista



Fonte: Autores, 2018.

Quando o item a ser removido não se encontra no início da lista, temos que realizar os passos a seguir:

1-localização do elemento a ser excluído. Além de localizar, é necessário atualizar o ponteiro do item anterior, ptAnt. Sendo assim, usamos um ponteiro auxiliar, ptAux, para procurar o elemento a ser removido e o ptAnt para guardar o elemento anterior na lista encadeada.

2- posicionamento dos ponteiros: Desta forma, ao encontrar o elemento a ser removido (Info 2), temos um apontador para o item anterior a ele (ptAnt) e outro para o próprio item (ptAux).

3-excluir o elemento: de posse destes apontadores, temos que ajustar o campo prox do elemento anterior ao removido: agora este campo tem que apontar para o elemento posterior ao removido. Sendo assim, ptAnt->prox aponta para ptAux->prox. Em seguida, remove-se o elemento.

4- atualizar a lista: ao final, temos uma lista encadeada com um item a menos e com todos os apontadores ajustados.

Essa sequência de passos é apresentada na Figura 30.

Figura 30- Código para a função excluir

```
67
68
69  /*
70   * Exclui um elemento da lista
71   */
72  listaEncadeada* excluir(listaEncadeada* iLista){
73      listaEncadeada* ptAnt = NULL;//ponteiro para o elemento anterior
74      listaEncadeada* ptAux = iLista;//ponteiro para percorrer a lista
75      int elemento;
76      printf("\n\nDigite um numero para ser excluido: ");
77      scanf("%d",&elemento);
78      /*Passo 1
79       * Localizar o elemento para ser excluido na lista
80       * guardando em ptAnt o endereço do nó anterior
81       */
82      while(ptAux!=NULL && ptAux->info != elemento){
83          ptAnt = ptAux;
84          ptAux = ptAux->prox;
85      }
86      /*Verifica se o elmento foi encontrado.Caso não encontre,
87       *retorna o iLista, que é uma lista original
88       */
89      if(ptAux == NULL){
90          return iLista;
91      }
92      /*Passo 2
93       * Exclui o primeiro elemento da lista
94       */
95      if(ptAnt == NULL){
96          iLista = ptAux->prox;
97      }else{
98          /*Exclui o elemento do meio até o último na lista*/
99          ptAnt->prox = ptAux->prox;
100     }
101     /*Passo 3*/
102     free(ptAux);//Libera o espaço de memória a ser excluido
103     /*Passo 4*/
104     return iLista;//retorna a lista
105 }
```

Fonte: Autores, 2018.

2.1.5 Destrói toda a lista

Antes de encerrar o programa temos que fazer a liberação dos espaços de memória alocados. A função apresentada na Figura 31 executa esta tarefa, ela destrói toda a lista, liberando os espaços de memória alocada em cada um dos blocos. Na linha 109 é criado um ponteiro ptAux, que aponta para o início da lista. Nas linhas 110 a 115 a lista é varrida e os blocos de memória alocados são liberados. Por fim, na linha 116 ela retorna NULL, pois não existe nenhum elemento na lista, e ela está vazia.

Figura 31- Código da função destrói.

```
106
107   /*Destroi a lista, liberando todos os elementos alocados*/
108   listaEncadeada* destroi(listaEncadeada* iLista){
109       listaEncadeada* ptAux = iLista;
110       while(ptAux != NULL){
111           //Guarda referência para o próximo ponteiro
112           listaEncadeada* ptTemp = ptAux->prox;
113           free(ptAux);//Libera a memória apontada por ptAux
114           ptAux=ptTemp;
115       }
116       return NULL;
117   }
118
```

Fonte: Autores, 2018.

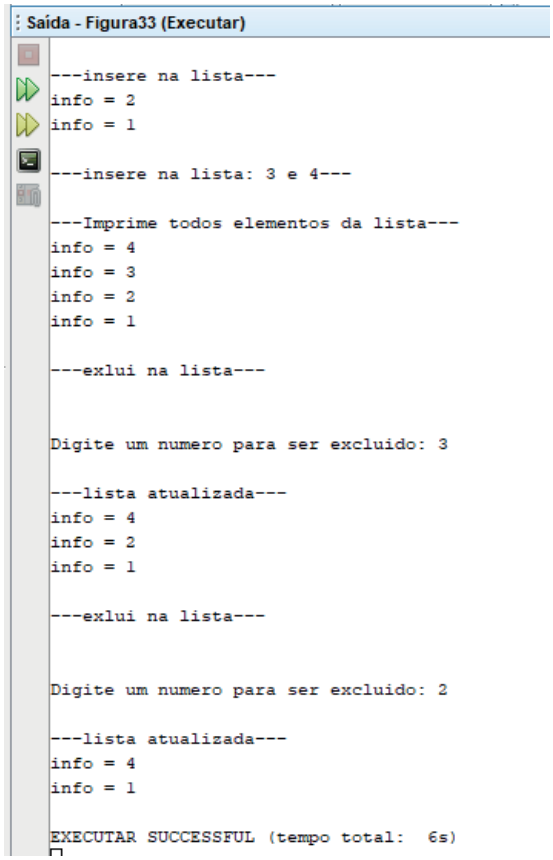
A função main, que ilustra a utilização das funções da lista encadeada é mostrada na Figura 32. É importante ressaltar que, a cada chamada da função insere ou excluir, o ptInicio é atualizado de acordo com a necessidade de cada função. A Figura 33 apresenta o resultado da execução do código da Figura 32.

Figura 32- Código da função main

```
118
119
120   int main(int argc, char** argv) {
121       listaEncadeada* ptInicio=NULL;
122       printf("\n---insere na lista---\n");
123       ptInicio=insere(ptInicio,1);//insere na lista o elemento 1
124       ptInicio=insere(ptInicio,2);//insere na lista o elemento 2
125       imprime(ptInicio);
126       printf("\n---insere na lista: 3 e 4---\n");
127       ptInicio=insere(ptInicio,3);//insere na lista o elemento 3
128       ptInicio=insere(ptInicio,4);//insere na lista o elemento 4
129       printf("\n---Imprime todos elementos da lista---\n");
130       imprime(ptInicio);
131       printf("\n---exlui na lista---\n");
132       ptInicio = excluir(ptInicio);
133       printf("\n---lista atualizada---\n");
134       imprime(ptInicio);
135       printf("\n---exlui na lista---\n");
136       ptInicio = excluir(ptInicio);
137       printf("\n---lista atualizada---\n");
138       imprime(ptInicio);
139
140       return 0;
141   }
```

Fonte: Autores, 2018.

Figura 33- Resultado da execução do programa da Figura 32



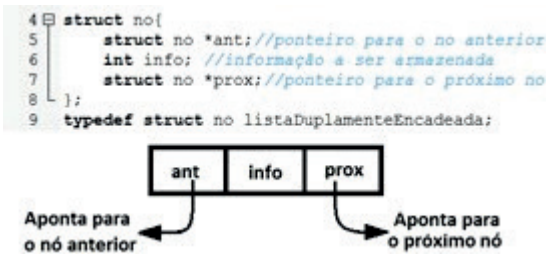
Fonte: Autores, 2018.

2.2
LISTAS DUPLAMENTE ENCADEADAS

Uma lista duplamente encadeada é aquela em que cada nó possui duas autorreferências, uma para o nó anterior e a outra para o próximo nó (ZIVIANI, 2007). Sendo assim, cada nó é gerado, além da informação que irá armazenar (seu conteúdo), com estas duas referências. Isso permite o caminhamento nos dois sentidos da lista duplamente encadeada, do início para o fim e, do fim para o início, diferentemente, da lista encadeada, que só permite o caminhamento em um único sentido.

As aplicações das listas duplamente encadeadas são semelhantes às das listas encadeadas, sendo que as listas duplamente encadeadas são mais apropriadas quando há a necessidade de caminhamento em ambos os sentidos. A Figura 34 apresenta a estrutura de lista duplamente encadeada auto referenciada, com três campos: um inteiro, um ponteiro prox, que aponta para o próximo nó do mesmo tipo e, o outro, um ponteiro ant, que aponta para o nó anterior do mesmo tipo.

Figura 34 - Estrutura e representação gráfica de um nó na lista duplamente encadeada



Fonte: Autores, 2018.

Ao construir uma lista duplamente encadeada, temos que realizar algumas operações com estas informações, abaixo estão as principais:

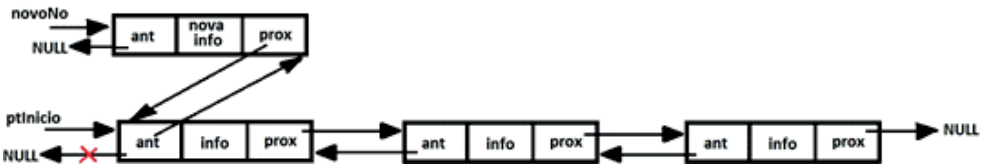
1 Inserção: insere um novo dado no início da lista;
2 Impressão: visita todos os nós da lista imprimindo o seu campo info (seu conteúdo);
3 Busca: realiza uma busca sequencial em toda a lista, verificando a existência de um determinado elemento na mesma;
4 Exclusão: remove um determinado elemento da lista.

A seguir são demonstradas e comentadas as operações de inserção e exclusão, por meio de programas modelos e representações gráficas, as demais funções podem ser iguais à lista encadeada, apresentadas anteriormente.

2.2.1Inserção no início da lista

A inserção de um elemento na lista perpassa por dois momentos, o primeiro é quando a lista está vazia, ou seja, ainda não foi inserido nenhum elemento. O segundo é quando a lista tem um ou mais elementos. Em ambos os casos a inserção, neste exemplo, sempre se dará no início da lista. Graficamente a inserção de um novo elemento é representada na Figura 35.

Figura 35 - Inserção no início da lista



Fonte: Autores, 2018.

O programa que executa a inserção no início da lista é mostrado na Figura 36. Ao criar um novo nó, na linha 37 e 38, na sequência é atribuída a informação a ser armazenada, no campo info, na linha 39, e o campo prox, na linha 40, é apontado para o início da lista. Para atualizar o campo ant são analisadas duas condições: se o elemento a ser inserido é o primeiro da lista, conforme linha 42, o campo novo-No->ant é apontado para NULL. Se a lista já possui um ou mais elementos, o campo iLista->ant é apontado para o novoNo, que foi gerado anteriormente na linha 37. E por último, na linha 47 é retornado o endereço no novoNo gerado, finalizando, desta forma, as atualizações de endereço de ponteiros, tanto o campo prox, quanto o campo ant foram atualizados.

Figura 36 - Código da função insere

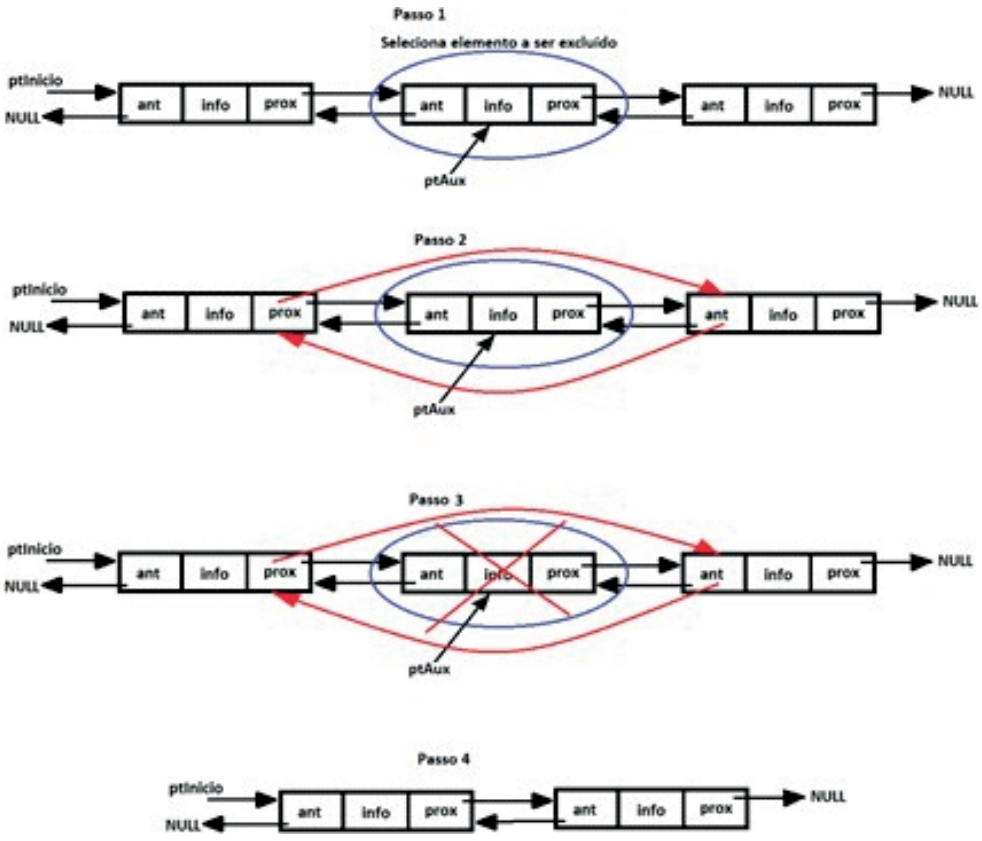
```
30  /*
31   * Insere um no início da lista e um ponteiro que armazena o endereço do inicio
32   * da lista
33   * iLista: ponteiro para o início da lista
34   * info: informação a ser armazenada no novo nó
35   */
36  listaDuplamenteEncadeada* insere(listaDuplamenteEncadeada* iLista, int info){
37      listaDuplamenteEncadeada* novoNo = (listaDuplamenteEncadeada*)
38              malloc(sizeof(listaDuplamenteEncadeada));
39      novoNo->info = info;
40      novoNo->prox =iLista;
41      //Insere o primeiro elemento da lista(iLista igual a NULL
42      if(iLista==NULL){
43          novoNo->ant=NULL;
44      }else{//Insre os demais elementos da lista
45          iLista->ant = novoNo;
46      }
47      return novoNo;
48  }
```

Fonte: Autores, 2018.

2.2.2 Exclusão de um elemento da lista

A função de exclusão apaga um nó da lista duplamente encadeada em qualquer posição e, ajusta os endereços de ponteiros, quando for necessário, para o início da lista, do nó anterior e para o próximo nó. A remoção de um elemento na lista duplamente encadeada é um pouco mais trabalhosa quando se compara com a lista encadeada. Na duplamente encadeada temos dois campos autorreferenciáveis, o ant e o prox, que permitem a utilização de um único ponteiro para guardar o endereço do nó a ser removido, chamado ptAux. Assim, a partir deste, é possível fazer as demais atualizações de endereços envolvendo os ponteiros. Este fato só é possível por conta dos dois campos ponteiros, o ant e o prox, que mantém um rastro de antes e depois, em todos os nós da lista. Conforme mostra a Figura 37, a exclusão é composta por 4 passos: 1-possicionar o ponteiro ptAux no elemento a ser removido, 2-Fazer as devidas atualizações de endereços de ponteiros, 3-liberar a memória e 4-atualizar a lista.

Figura 37- Exclusão de um elemento da lista



Fonte: Autores, 2018.

O programa que executa a exclusão de um nó em qualquer posição da lista é mostrado na Figura 38. Inicialmente foi criado um ponteiro ptAux para percorrer toda a lista, que aponta para o início da mesma, na linha 79. Nas linhas 81 e 82 é feita uma solicitação ao usuário para que ele digite o elemento a ser removido. Após saber qual elemento deve ser removido, temos que localizá-lo dentro da lista, essa

tarefa é realizada nas linhas 86 a 88. Para excluir o referido elemento temos que observar se ele é o primeiro da lista, caso seja, o início da lista, o iLista, passa ser o endereço do prox, por meio da instrução iLista = ptAux->prox, apresentada na linha 97. Caso o elemento não seja o primeiro, tem-se que atualizar o ponteiro prox, por meio da instrução ptAux->ant->prox = ptAux->prox, conforme mostra na linha 100. Já o ponteiro ant é atualizado conforme o código ptAux->prox->ant = ptAux->ant, na linha 105, que independe da quantidade de elementos existentes na lista.

Figura 38 - Código da função exclusão

```
77   /*Exclui um elemento da lista */
78   listaDuplamenteEncadeada* excluir(listaDuplamenteEncadeada* iLista){
79       listaDuplamenteEncadeada* ptAux = iLista;//ponteiro para percorrer a lista
80       int elemento;
81       printf("\n\nDigite um numero para ser excluido: ");
82       scanf("%d",&elemento);
83       /*Passo 1
84        * Localizar o elemento para ser excluido na lista
85        * guardando em ptAnt o endereço do nó anterior */
86       while(ptAux!=NULL && ptAux->info != elemento){
87           ptAux = ptAux->prox;
88       }
89       /*Caso não encontre o elementro procurado
90        *retorna o iLista, que a lista original */
91       if(ptAux == NULL){
92           return iLista;
93       }
94       /*Passo 2
95        * Exclui o primeiro elemento e atualiza a lista */
96       if(iLista == ptAux){
97           iLista = ptAux->prox;
98           //Exclui da calda da lista e atualiza o ponteiro do campo prox
99       }else{
100          ptAux->ant->prox = ptAux->prox;
101      }
102
103      //Atualiza o ponteiro ant
104      if(ptAux->prox != NULL){
105          ptAux->prox->ant=ptAux->ant;
106      }
107      /*Passo 3*/
108      free(ptAux);//Libera o espaço de memória a ser excluido
109      /*Passo 4*/
110      return iLista;//retorna a lista
111  }
```

Fonte: Autores, 2018.

A função *main*, que ilustra as chamadas das funções da lista duplamente encadeada, é mostrada na Figura 39. Assim como na lista encadeada, é importante ressaltar que a cada chamada da função insere ou excluir o ptInicio deve ser atualizado.

Figura 39 - Código da função *main*

```
126  int main(int argc, char** argv) {
127      listaDuplamenteEncadeada* ptInicio=NULL;
128      printf("\n---insere na lista Duplamente Encadeada ---\n");
129      ptInicio=insere(ptInicio,1);//insere na lista o elemento 1
130      ptInicio=insere(ptInicio,2);//insere na lista o elemento 2
131      imprime(ptInicio);
132      printf("\n---insere na lista: 3 e 4---\n");
133      ptInicio=insere(ptInicio,3);//insere na lista o elemento 3
134      ptInicio=insere(ptInicio,4);//insere na lista o elemento 4
135      printf("\n---Imprime todos elementos da lista---\n");
136      imprime(ptInicio);
137      printf("\n---exlui na lista---\n");
138      ptInicio = excluir(ptInicio);
139      printf("\n---lista atualizada---\n");
140      imprime(ptInicio);
141      printf("\n---exlui na lista---\n");
142      ptInicio = excluir(ptInicio);
143      printf("\n---lista atualizada---\n");
144      imprime(ptInicio);
145
146      return 0;
147  }
```

Fonte: Autores, 2018.

Existem diversas maneiras de fazer uma lista, por exemplo, outra maneira seria usar tipos abstratos de dados, que iremos estudar na próxima unidade. Assista às videoaulas sobre o assunto antes de prosseguir.

INTERATIVIDADE:

Lista Dinâmica Encadeada - Parte 1
https://www.youtube.com/watch?v=0BDMqra4D94

Lista Dinâmica Encadeada Parte 2 - Implementação
https://www.youtube.com/watch?v=wfC61zUVaos

Lista Dinâmica Encadeada Parte 3 - Informações
https://www.youtube.com/watch?v=WvmBhiQjPZ0

Inserção na Lista Dinâmica
https://www.youtube.com/watch?v=fNP1GHLLKuY

Remoção na Lista Dinâmica
https://www.youtube.com/watch?v=67KZx_Rcfgw

Atividades – Unidade 2

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Construa uma lista duplamente encadeada, de acordo com a Figura 40.

Figura 40 -Lista duplamente encadeada completa

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

using namespace std;

/*
 *
 */

struct no{
    struct no *ant;//ponteiro para o nó anterior
    int info;//informação a ser armazenada
    struct no *prox;//ponteiro para o próximo nó
};
typedef struct no listaDuplamenteEncadeada;
```

```
/*
 * Insere um no início da lista e um ponteiro que armazena o endereço do início
 * da lista
 * iLista: ponteiro para o início da lista
 * info: informação a ser armazenada no novo nó
 */
listaDuplamenteEncadeada* insere(listaDuplamenteEncadeada* iLista, int info){
    listaDuplamenteEncadeada* novoNo = (listaDuplamenteEncadeada*)
            malloc(sizeof(listaDuplamenteEncadeada));
    novoNo->info = info;
    novoNo->prox =iLista;
    //Insere o primeiro elemento da lista(iLista igual a NULL
    if(iLista==NULL){
        novoNo->ant=NULL;
    }else{//Insre os demais elementos da lista
        iLista->ant = novoNo;
    }
    return novoNo;
}
```

```
/*
 * Imprime o campo info dos nós que foram armazenados na lista
 * iLista: ponteiro para o início da lista
 */
void imprime (listaDuplamenteEncadeada* iLista){
    listaDuplamenteEncadeada* ptAux; // ponteiro auxiliar para percorrer a lista
    for(ptAux = iLista; ptAux != NULL; ptAux=ptAux->prox){
        printf("info = %d \n",ptAux->info);//para funcionar tem q incluir stdio.h
    }
}

/*Busca um elemento na lista e retorna o endereço do nó que
 * contém o elemento. Se tal nó não existe, a função devolve NULL
 *  iLista: ponteiro para o início da lista
 *  elemento: informação a ser buscada na lista
 */
listaDuplamenteEncadeada* busca(listaDuplamenteEncadeada* iLista, int elemento){
    listaDuplamenteEncadeada* ptAux;
    for(ptAux = iLista; ptAux != NULL; ptAux=ptAux->prox){
        if(ptAux->info == elemento){
            return ptAux;
        }
    }
    return NULL;//Não encntrou o elemento
}
```

```
75
76
77      /*Exclui um elemento da lista */
78      listaDuplamenteEncadeada* excluir(listaDuplamenteEncadeada* iLista){
79          listaDuplamenteEncadeada* ptAux = iLista;//ponteiro para percorrer a lista
80          int elemento;
81          printf("\n\nDigite um numero para ser excluido: ");
82          scanf("%d",&elemento);
83          /*Passo 1
84           * Localizar o elemento para ser excluido na lista
85           * guardando em ptAnt o endereço do nó anterior */
86          while(ptAux!=NULL && ptAux->info != elemento){
87              ptAux = ptAux->prox;
88          }
103         //Atualiza o ponteiro ant
104         if(ptAux->prox != NULL){
105             ptAux->prox->ant=ptAux->ant;
106         }
107         /*Passo 3*/
108         free(ptAux);//Libera o espaço de memória a ser excluido
109         /*Passo 4*/
110         return iLista;//retorna a lista
111     }
112
 99         }else{
100             ptAux->ant->prox = ptAux->prox;
101         }
102
103         //Atualiza o ponteiro ant
104         if(ptAux->prox != NULL){
105             ptAux->prox->ant=ptAux->ant;
106         }
107         /*Passo 3*/
108         free(ptAux);//Libera o espaço de memória a ser excluido
109         /*Passo 4*/
110         return iLista;//retorna a lista
111     }
```

```
113     /*Destroi a lista, liberando todos os elementos alocados*/
114     listaDuplamenteEncadeada* destroi(listaDuplamenteEncadeada* iLista){
115         listaDuplamenteEncadeada* ptAux = iLista;
116         while(ptAux != NULL){
117             //Guarda referência para o próximo ponteiro
118             listaDuplamenteEncadeada* ptTemp = ptAux->prox;
119             free(ptAux);//Libera a memória apontada por ptAux
120             ptAux=ptTemp;
121         }
122         return NULL;
123     }
124
```

```
125
126     int main(int argc, char** argv) {
127         listaDuplamenteEncadeada* ptInicio=NULL;
128         printf("\n---insere na lista Duplamente Encadeada ---\n");
129         ptInicio=insere(ptInicio,1);//insere na lista o elemento 1
130         ptInicio=insere(ptInicio,2);//insere na lista o elemento 2
131         imprime(ptInicio);
132         printf("\n---insere na lista: 3 e 4---\n");
133         ptInicio=insere(ptInicio,3);//insere na lista o elemento 3
134         ptInicio=insere(ptInicio,4);//insere na lista o elemento 4
135         printf("\n---Imprime todos elementos da lista---\n");
136         imprime(ptInicio);
137         printf("\n---exlui na lista---\n");
138         ptInicio = excluir(ptInicio);
139         printf("\n---lista atualizada---\n");
140         imprime(ptInicio);
141         printf("\n---exlui na lista---\n");
142         ptInicio = excluir(ptInicio);
143         printf("\n---lista atualizada---\n");
144         imprime(ptInicio);
145
146         return 0;
147     }
```

Fonte: Autores, 2018.

2) Construa uma lista encadeada, que armazene o nome dos alunos, e que faça as seguintes operações:

1–exibir lista
2–inserir um elemento
3 –remover um elemento
4 –exibir a posição de um elemento

3

PILHAS E FILAS

INTRODUÇÃO

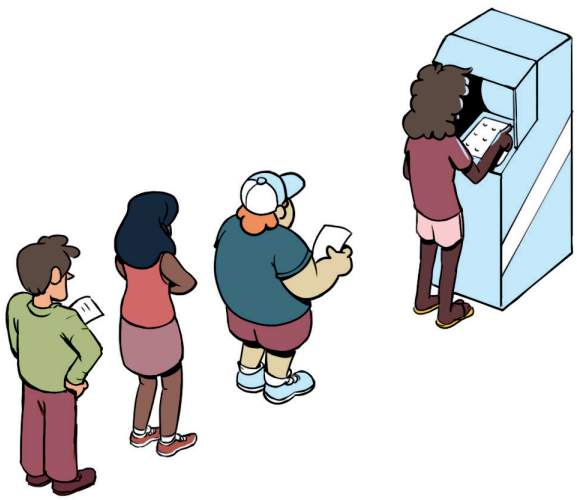
Muitos dos problemas a serem resolvidos em nosso cotidiano envolvem a representação e a manipulação de sequência ordenada de objetos, como por exemplo, a representação de uma seleção de livros empilhados para leitura e uma fila de pessoas em um caixa de um banco, conforme apresentado nas figuras 41 e 42.

Figura 41- Pilha de livros

Fonte: NTE/UFSM.

Suponha que você selecionou 4 livros de estrutura de dados para estudar e os empilhou na seguinte ordem: livro verde, depois o amarelo, vermelho e por fim, o azul, conforme apresentado na Figura 41. Ao iniciar os estudos sobre a nossa disciplina qual livro você pode retirar primeiro da pilha? Seria o livro verde? Com certeza não, caso ele fosse retirado a pilha desmoronaria, não queremos que isso aconteça. Sendo assim a ordem correta é retirar primeiro o livro azul, depois o vermelho, e o amarelo até chegar ao verde, por exemplo. Vamos utilizar essas sequências para construir os programas referentes à estrutura de dados pilha, logo, essa mesma disciplina de inserção e remoção deve ser preservada, ou seja, o último elemento a ser empilhado, que é o mais recente na pilha, será o primeiro a ser retirado.

Figura 41- Fila de pessoas em um banco

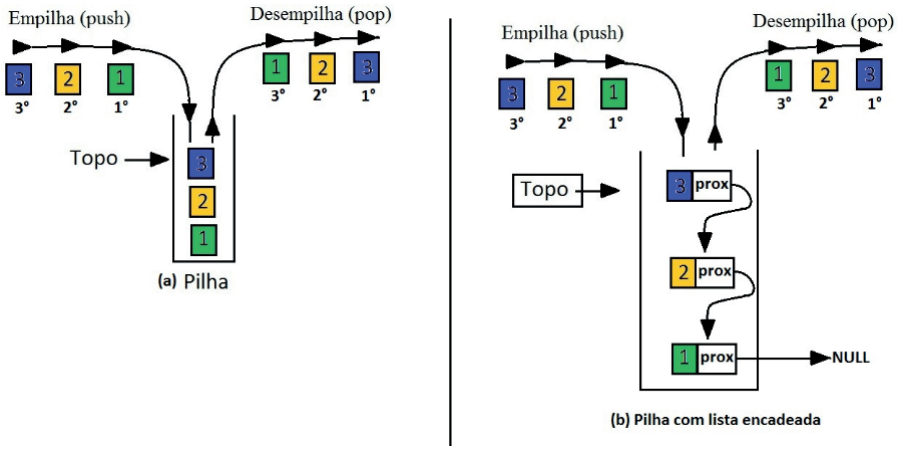


Fonte: NTE/UFSM.

Agora, conforme mostra a Figura 42, vamos supor que o seu problema a ser resolvido se resume no pagamento de uma conta de energia elétrica, para isso, vá a uma agência e entre no final da fila do caixa. Analisando a fila, queremos saber, quem irá ser atendido primeiro? Quem está no início o ou no final da fila? As filas obedecem a ordem de chegada, sendo assim quem estiver no início será atendido primeiro. Ao construir o seu programa que implemente uma fila, naturalmente, a restrição de chegada deve ser mantida, logo, o primeiro elemento a chegar será o primeiro a sair.

No mundo computacional dizemos que uma sequência ordenada de objetos é uma lista linear. Portanto, pilhas e filas são listas lineares com restrições de inserção, remoção e consulta dos seus elementos. A seguir vamos abordar a construção de tipos abstratos de dados e depois detalhar a implementação das estruturas de dados pilha e fila.

Figura 43: Exemplo de Pilha



Fonte: NTE, 2018.

As pilhas podem ser construídas de diversas maneiras. Dentre elas, citamos a implementação por meio de vetores, quando sabemos o tamanho exato da pilha, e a implementação com alocação dinâmica, quando queremos armazenar um tamanho indefinido de informações. Em nossos estudos vamos adotar alocação dinâmica. Em qualquer das abordagens, as pilhas apresentam duas operações principais, a inserção e a remoção, que por razões históricas, são normalmente chamadas de push e pop, respectivamente. A Figura 43, letra (b) mostra a representação gráfica de uma pilha alocada dinamicamente por meio de lista encadeada. Para implementar a nossa pilha vamos construir quatro operações: push (inserir ou empilhar), pop (desempilhar ou excluir), criar uma pilha vazia, imprimir toda a pilha iniciando do topo e liberar todos os blocos alocados de memória.

3.1
TIPOS ABSTRATO DE DADOS

Um Tipo Abstrato de Dados (TAD) é uma abstração da realidade, mediante a definição de um conjunto de dados que a representa e quais operações serão realizadas sobre estes dados (TENENBAUM, 1995). Desta forma, a ideia básica do TAD é desvincular o tipo de dado (valores e operações) de sua implementação. Ao usar os TADs, temos algumas vantagens, provenientes desta desvinculação, são elas:

1. Integridade dos dados;
2. Facilidade de manutenção;
3. Reutilização.

Um TAD pode ser definido matematicamente pelo par (V,O), onde V é um conjunto de dados, ou valores e, O um conjunto de operações sobre estes valores.

Em geral, quando usamos o conceito de tipo abstrato de dados, dividimos a programação em duas etapas:

- Especificação, ou a interface (em Linguagem C, são arquivos *.h) e
- Operação (em C, são arquivos *.c).

Na interface, apenas as especificações das operações são definidas abstratamente, não sendo permitido o acesso direto aos dados, ou seja, o usuário só tem acesso às assinaturas das operações. Por exemplo, vamos construir um programa que trabalhe com listas de números inteiros e com as operações de inclusão e busca. Neste tipo abstrato de dados o usuário nunca poderia trocar a posição de dois elementos na lista, pois a ele só é permitido inserir ou procurar por um elemento. As características da operação permanecem inacessíveis, ou seja, o usuário não consegue saber, por exemplo, se a lista é dinâmica ou estática. Desta forma, implementar um TAD em uma linguagem de programação é encontrar uma forma de representá-lo utilizando esta linguagem, levando em consideração os tipos nativos e suas operações suportadas pelo computador. Em C, a implementação destes TADs se dá, basicamente, por meio de ponteiros e estruturas. Sendo assim, a implementação de um TAD genérico para uma lista linear deve possuir os seguintes itens (LAFORE, 2004):

- Interface: São as variáveis definidas na estrutura. Elas representam o tipo de dados a serem armazenados na lista, por exemplo, dados do tipo inteiro ou os dados para armazenar os endereços, que são do tipo ponteiro, além das assinaturas das operações.
- Operações: Implementa todo o código das ações realizadas na lista, por exemplo: insere, remove, consulta, altera e destrói a lista.

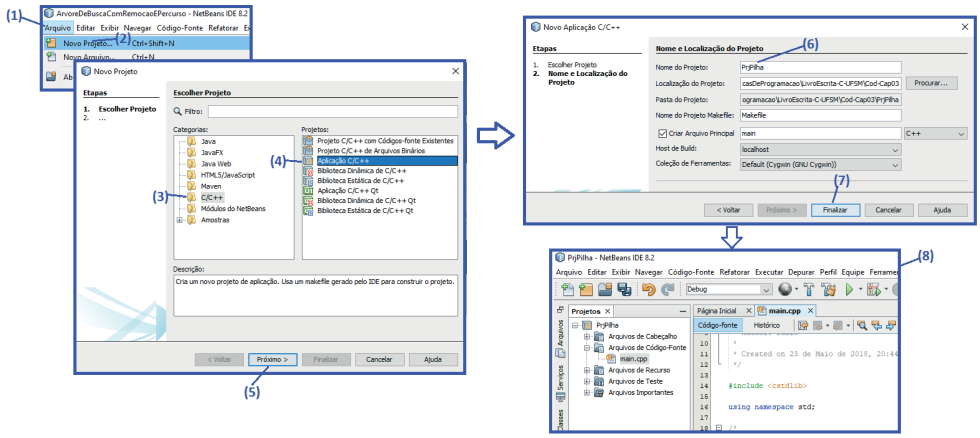
3.2
PILHAS

A estrutura pilha é um caso particular de lista linear, pois esta apresenta disciplina de acesso em todas as suas operações. Desta forma as inserções, remoções e consultas são realizadas em um único extremo, o topo. Assim, quando um novo elemento é inserido na pilha, ele passa a ser o elemento do topo e, além disso, o único elemento que pode ser removido da pilha é aquele que se encontra no topo. Portanto, o critério de utilização adotado em cada uma das operações é conhecido como LIFO (*Last In First Out*), conforme apresentado na figura 43, letra (a), o último elemento a ser empilhado (inserido), é o primeiro a ser desempilhado (removido). Logo, as operações devem ser feitas no topo da pilha e, na ordem inversa a sua inserção.

3.2.1 TAD para pilha em Linguagem C

Vamos implementar o TAD no NetBeans 8.2 (Ambiente de desenvolvimento que você já utilizou anteriormente no Curso de Licenciatura em Computação), com o plug-ing de C/C++ devidamente configurado. Para criar um novo projeto, em nosso exemplo chamado PrjPilha, siga a sequência de passos de 1 a 8, conforme mostra a Figura 44.

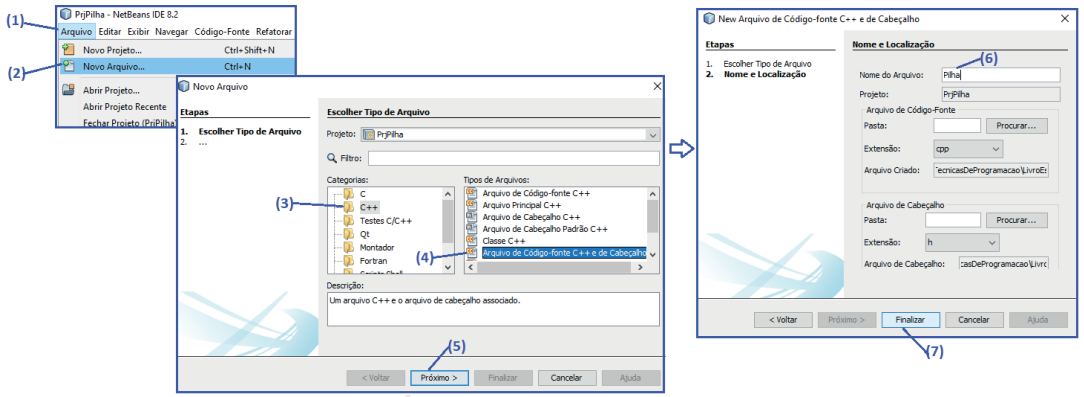
Figura 44: Criação do projeto PrjPilha no NetBeans



Fonte: Autores, 2018.

Ao finalizar os passos de 1 a 7, descritos na Figura 44, o resultado é mostrado no passo 8. Veja que o *NetBeans* já cria um arquivo com o nome "main.cpp". Agora temos que inserir mais dois arquivos, conforme sequência de passos apresentados na Figura 45.

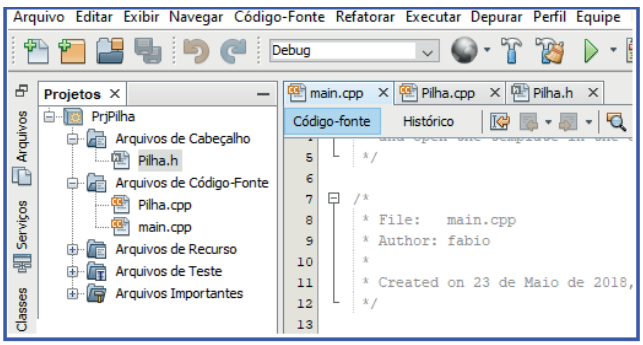
Figura 45 - Criação de novos arquivos no projeto PrjPilha



Fonte: Autores, 2018.

Ao final teremos 3 arquivos, conforme apresentado na Figura 46. A composição do projeto obedece a seguinte ordem: a primeira parte do projeto é a construção da interface da pilha, depois as operações e por fim, a função principal que fará uso de todas as operações implementadas.

Figura 46 - Criação dos arquivos *Plha.cpp* e *Pilha.h*



Fonte: Autores, 2018.

3.2.1.1 Interface

Para construir a interface clique sobre o arquivo "Pilha.h" e digite o código apresentado na Figura 47. Após finalizar a digitação salve o arquivo. Este arquivo é composto somente pela assinatura das funções (cabeçalhos ou *headers*) e as variáveis definidas para as estruturas. Portanto, ele tem por objetivo demonstrar a forma correta das chamadas e a existência das operações.

Figura 47 - Interface da fila no arquivo Pilha.h.

```
1   /*...5 linhas */
6
7   /*...6 linhas */
13
14  #ifndef PILHA_H
15  #define PILHA_H
16
17  //Esrutura paa criar uma lista encadeada
18  typedef struct no listaEncadeada;
19
20  //Estrutura para controlar a Pilha
21  typedef struct pilha controlePilha;
22
23  //Cria uma pilha, inicialmente vazia
24  controlePilha* cria(void);
25
26  //Insere os dados no topo da Pilha
27  void push(controlePilha* ,int info);
28
29  //Retira e exclui um elemento do topo da Pilha
30  int pop(controlePilha* ptPilha);
31
32  //Libera todos os blocos de memória alocados
33  void libera(controlePilha* ptPilha);
34
35  //Imprime toda a Pilha, do topo ao fim
36  void imprime(controlePilha* ptPilha);
37
38  #endif /* PILHA_H */
```

Fonte: Autores, 2018.

Conforme observado na Figura 47, nas linhas 18 e 21 são declaradas as variáveis listaEncadeada e controlePilha: a primeira é do tipo nó e a outro é do tipo pilha, ambas são estruturas. A função cria, na linha 24, um novo nó dinamicamente e, inicializa os seus campos. Ao final ela retorna o endereço do nó criado. Na linha 27 a função *push* recebe os parâmetros ptPilha e info. O ptPilha é um ponteiro definido para estrutura que controla o topo da pilha e o info é a informação a ser armazenada em cada nó. A função *pop*, na linha 30, remove um elemento do topo da pilha. Para tal, ela recebe como parâmetro o ponteiro ptPilha. A função libera, na linha 33, remove todos os blocos de memória alocados para a pilha, recebendo como parâmetro o ponteiro de controle da pilha, o ptPilha. Na linha 36, a função imprime percorre toda a pilha seguindo a ordem do topo para o final; ela recebe o como parâmetro o ponteiro ptPilha.

3.2.1.2 Implementação das operações

A seguir serão implementadas as operações da pilha, o que consiste em construir as estruturas e as funções da interface. Para construí-las você deve digitar o código abaixo, como mostra a Figura 48, no arquivo *Pilha.cpp* e salvar.

Figura 48 - Função cria, no arquivo Pilha.cpp

```
1  /*...5 linhas */

7  /*...6 linhas */

13
14 #include "Pilha.h"
15 #include <stdio.h>
16 #include <stdlib.h>
17
18 //Estrutura para criar uma lista encadeada
19 struct no{
20     int info;
21     struct no* prox;
22 };
23
24 //Estutura paa controlar a Pilha
25 struct pilha{
26     listaEncadeada* topo;
27 };
28
29 //Cria uma Pilha vazia
30 controlePilha* cria(void){
31     controlePilha* ptPilha = (controlePilha*)malloc(sizeof(controlePilha));
32     ptPilha->topo = NULL;
33     return ptPilha;
34 }
```

Fonte: Autores, 2018.

A Figura 48, mostra as inclusões de arquivos *.h, a implementação das estruturas e a primeira função do projeto. Além de incluir os arquivos rotineiros, das linhas 15 e 16, você tem que incluir, ainda, o arquivo da interface criado para pilha, o Pilha.h, conforme mostra a linha 14. Nas linhas de 19 a 22 foi implementada a estrutura para o nó. Essa estrutura tem a responsabilidade de armazenar a informação e o endereço do próximo nó. Nas linhas 25 a 27 foi criada a estrutura para impor a disciplina de acesso da pilha. Logo, ela deve conter o endereço do elemento que está no topo da pilha. A função cria, na linha 30 a 34, aloca um espaço na memória para o controle da pilha, inicializando o ponteiro topo com NULL e retorna o ponteiro ptPilha.

Conforme apresentado na Figura 49, a função *push*, tem por objetivo inserir um novo elemento no topo da pilha. Para entendê-la vamos começar analisando a linha 38. Nela é alocada memória para um novo nó da lista encadeada, inserindo a informação no campo *novoNo->info*, na linha 39, e aponta o novo nó criado para o endereço do topo, com o código *novoNo->prox = ptPilha->topo*. Na linha 40 atualiza-se o elemento do topo para o endereço do novoNo, na linha 41. Desta forma, todo elemento a ser inserido sempre será inserido no topo.

Já função pop, apresentada na Fgura 49, tem a finalidade de excluir um elemento do topo. Para isso, na linha 46, foi declarado um ponteiro *ptAux*, que auxilia na exclusão dos elementos. Na linha 54, uma variável do tipo inteiro, chamada *info*, foi declarada para armazenar o valor do elemento a ser excluído e, ao final da função, o valor excluído é retornado, na linha 56, para a função chamadora.

Antes de excluir um elemento da pilha temos que comprovar se a pilha não está vazia. Esse teste é realizado na linha 48. Caso a pilha esteja vazia o programa é finalizado, na linha 50. Se existe pelo menos um elemento na pilha, então o ponteiro *ptAux* é posicionado para o topo da pilha, conforme representado na linha 52, e a informação contida em *ptAux->info* é atribuída à variável info, na linha 53. Na linha 54 acontece a atualização do endereço do topo, uma vez que iremos remover o atual elemento que se encontra no topo, o próximo passo é posicionar ptPiha para o endereço de *ptAux->prox*, por tanto o ponteiro *ptPiha->topo* passa a apontar para *ptAux->prox*. A remoção do topo atual é realizada na linha 55, por meio do comando *free(ptAux)*.

Figura 49 - Função push e pop, no arquivo Pilha.cpp

```
36  //Insere os dados no topo da Pilha
37  void push(controlePilha* ptPilha, int info){
38      listaEncadeada* novoNo = (listaEncadeada*)malloc(sizeof(listaEncadeada));
39      novoNo->info = info;
40      novoNo->prox=ptPilha->topo;
41      ptPilha->topo=novoNo;
42  }
43
44  //Retira e exlcui um elemento do topo da Pilha
45  int pop(controlePilha* ptPilha){
46      listaEncadeada* ptAux;
47      int info;
48      if(ptPilha->topo==NULL){//Verifica se a lista não estava vazia
49          printf("Pilha vazia, insira elementos");
50          exit(1);
51      }
52      ptAux = ptPilha ->topo;
53      info= ptAux->info;
54      ptPilha->topo = ptAux->prox;
55      free(ptAux);
56      return(info);
57  }
58
```

Fonte: Autores, 2018.

Conforme representado graficamente, na Figura 50, a função libera utiliza dois ponteiros auxiliares, o *ptAuxPer*, na linha 61, para percorrer toda a pilha, e libera os blocos de memória alocado, conforme linha 66, por meio da função *free(ptAuxPer)*. O ponteiro *ptAuxEnd*, na linha 62, tem a finalidade de guardar o endereço do próximo bloco de memória a ser removido, na linha 65, para depois atualizar o *ptAuxPer*, na linha 67.

Ainda na Figura 50, a função imprime varre toda a pilha na direção do topo para o fim. Para isso, na linha 76, é declarado o ptAux para percorrê-la. A este ponteiro, na linha 77, é atribuído o endereço do primeiro elemento da pilha, por meio da atribuição *ptAux=ptPilha->topo*. Na mesma linha é estabelecido também, o critério de parada, o laço irá repetir até que *ptAux* seja diferente de *NULL*, e por fim é atualizado o endereço para o próximo elemento da pilha, *ptAux = ptAux->prox*.

Figura 50 - Função libera e imprime, no arquivo *Pilha.cpp*

```
59  //Libera todos os blocos de memoria alocados
60  void libera(controlePilha* ptPilha){
61      listaEncadeada* ptAuxPer;//Ponteiro para percorrer a Pilha
62      listaEncadeada* ptAuxEnd;//Ponteiro para armaenar o endereço do prox nó
63      ptAuxPer = ptPilha->topo;
64      while(ptAuxPer != NULL){
65          ptAuxEnd = ptAuxPer->prox;
66          free(ptAuxPer);
67          ptAuxPer = ptAuxEnd;
68      }
69      free(ptPilha);
70      printf("\n Pilha Liberada!!!!");
71  }
72
73
74  //Imprime toda a Pilha, do topo ao fim
75  void imprime(controlePilha* ptPilha){
76      listaEncadeada* ptAux;
77      for(ptAux=ptPilha->topo;ptAux!=NULL; ptAux=ptAux->prox){
78          printf("\n info->%d",ptAux->info);
79      }
80  }
```

Fonte: Autores, 2018.

3.2.1.3 Exemplo de utilização do TAD

A Figura 51 mostra um exemplo de utilização do TAD definido para a pilha. Digite o código da respectiva figura no arquivo *main.cpp* e salve o código digitado. Ao digitar este código você irá construir as chamadas para as funções previamente definidas em *Pilha.h* e implementadas em *Pilha.cpp*.

Figura 51 - Função main, no arquivo *main.cpp*

```
1   /*...5 linhas */

7   /*...6 linhas */

14  #include <cstdlib>
15  #include <stdio.h>
16  #include <stdlib.h>
17  #include "Pilha.h"

19  using namespace std;

21  /*...3 linhas */
24  int main(int argc, char** argv) {
25      controlePilha* ptPilha = cria();
26      int numero, i;
27      printf("PUSH-Insere 5 numeros na pilha\n");
28      for(i=0;i<5;i++){
29          printf("Digite o numero (%d) a ser inserido na pilha: ",i);
30          scanf("%d",&numero);
31          push(ptPilha, numero);
32      }
```

```
33      printf("\n Imprime toda a lista");
34      imprime(ptPilha);
35      printf("\n\n POP-Exclui 3 numeros da pilha");
36      printf("\n Exclui o numero (%d) do topo da pilha",pop(ptPilha));
37      printf("\n Exclui o numero (%d) do topo da pilha",pop(ptPilha));
38      printf("\n Exclui o numero (%d) do topo da pilha",pop(ptPilha));
39      imprime(ptPilha);
40      libera(ptPilha);
41      return 0;
42  }
```

Fonte: Autores, 2018.

Para que o programa, exposto na Figura 51 funcione corretamente, faz-se necessário a inclusão do arquivo *Pilha.h*, conforme mostra a linha 17, pois este arquivo contém as definições das interfaces. Logo na sequência, na linha 25, é criada uma pilha vazia. Nas linhas de 28 a 32, são inseridos 5 elementos do tipo inteiro na pilha, por meio da função *push*. Na linha 34, é solicitada a impressão de toda a pilha e, nas linhas 36 a 38 é chamada a função pop para fazer a exclusão de três elementos, pegando sempre o que estiver no topo. Para verificar se os elementos foram excluídos corretamente, na linha 39, é solicitada novamente a impressão da pilha. Por fim, na linha 40, solicita-se a liberação dos blocos de memória alocados. O resultado da execução do arquivo *main.cpp* é apresentado na Figura 52.

Figura 52 - Resultado da execução do arquivo *main.cpp*

```
PUSH-Insere 5 numeros na pilha
Digite o numero (0) a ser inserido na pilha: 1
Digite o numero (1) a ser inserido na pilha: 2
Digite o numero (2) a ser inserido na pilha: 3
Digite o numero (3) a ser inserido na pilha: 4
Digite o numero (4) a ser inserido na pilha: 5

 Imprime toda a lista
 info->5
 info->4
 info->3
 info->2
 info->1

 POP-Exclui 3 numeros da pilha
 Exclui o numero (5) do topo da pilha
 Exclui o numero (4) do topo da pilha
 Exclui o numero (3) do topo da pilha

 Imprime toda a lista
 info->2
 info->1
 Pilha Liberada!!!!
EXECUTAR SUCCESSFUL (tempo total:  5s)
```

Fonte: Autores, 2018.

Analisando a Figura 52, verifica-se que foram inseridos números de 1 a 5, e primeiro número a ser retirado foi o número 5. Isso indica que ele estava no topo, ou seja, foi o último número a ser inserido. Logo, podemos concluir que o último número a entrar na pilha foi o primeiro a sair, por tanto, a disciplina de acesso foi implementada com sucesso. Vamos aprofundar o conhecimento sobre pilha assistindo às videoaulas e praticando o código explicado.

INTERATIVIDADE:
Assista a:
Pilha: Definição
https://www.youtube.com/watch?v=2RCrd7gOUMM

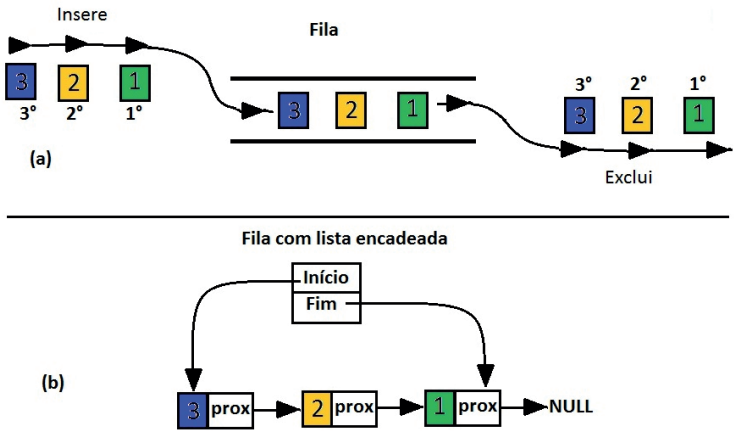
Pilha Dinâmica - Introdução
https://www.youtube.com/watch?v=9GGJH2sjOac

Pilha Dinâmica - Inserção e Remoção
https://www.youtube.com/watch?v=06sqFaB3gUo

3.3
FILAS

Outra estrutura de dados bastante usada em computação são as filas. Assim como as pilhas, são um caso particular de lista linear, embora com disciplina de acesso, nas suas operações, diferente das pilhas (CELES, 2004). Nas filas, as inserções são realizadas sempre no final e as remoções somente no início. Desta forma ela não permite acessar um elemento que não seja o primeiro. Esses critérios são conhecidos como FIFO - First In First Out, o primeiro elemento a entrar é o primeiro a sair, sendo assim, eles entram por um lado e saem por outro, conforme mostra a Figura 53, letra (a).

Figura 53 - Representação gráfica de uma fila



Fonte: Autores, 2018.

Rotineiramente as filas são utilizadas na computação para administrar recursos compartilhados, impondo uma prioridade por ordem de chegada, por exemplo, as filas de impressão, onde cada documento espera a sua vez para ser impresso de acordo com a ordem de chegada.
De modo análogo ao que implementamos com a estrutura de pilha, adotaremos a estratégia de implementação tendo como base a lista encadeada, conforme mostra a Figura 53, letra (b).

Para manipular as informações da fila vamos implementar as seguintes operações:

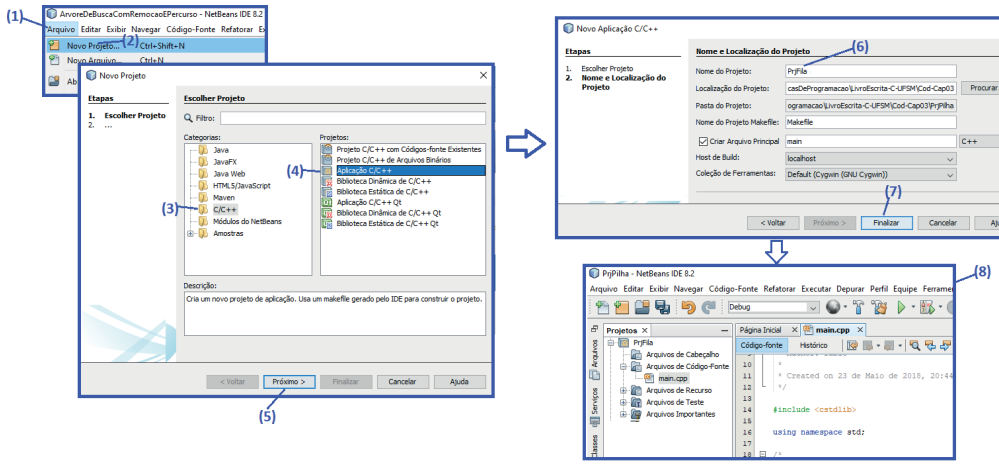
- criar uma estrutura de fila;
- inserir um elemento no fim;
- excluir o elemento do início;
- imprimir toda a lista começando do início; e
- liberar a fila.

Na sequência será apresentada a implementação do TAD para uma fila com estas operações.

3.3.1TAD para fila em Linguagem C

Inicialmente crie um novo projeto no *NetBeans*, chamado *PrjFila*, de acordo com a sequência de passos de 1 a 8, conforme mostra a Figura 54.

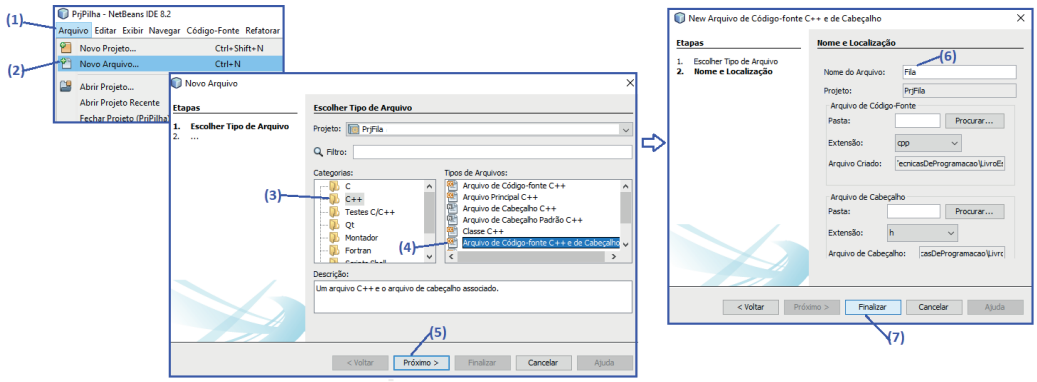
Figura 54 - Criação do projeto *PrjFila* no *NetBeans*



Fonte: Autores, 2018.

Ao finalizar os passos enumerados de 1 a 7, na Figura 54, o resultado das ações é apresentado no passo 8, relembrando que o *NetBeans* cria, junto com projeto, um arquivo com o nome main.cpp. Na sequência, vamos inserir dois novos arquivos, conforme mostra a Figura 55.

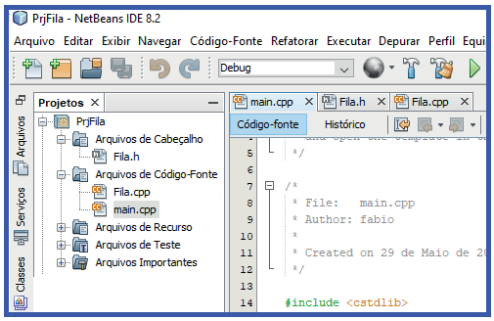
Figura 55 - Criação de novos arquivos no projeto *PrjFila*



Fonte: Autores, 2018.

Ao final dos passos enumerados de 1 a 7 na Figura 55, teremos um total de 3 arquivos no projeto, representado na Figura 56.

Figura 56 – Criação dos arquivos *Fila.cpp* e *Fila.h*



Fonte: Autores, 2018.

A seguir vamos, inicialmente, construir a interface da fila, com todas as operações já citadas. Depois estas operações serão implementadas e, por fim, faremos a implementação do código no arquivo *main.cpp* que fará uso da interface e das operações implementadas.

3.3.1.1 Interface

A interface deve ser implementada no arquivo *Fila.h*. Clique sobre ele e digite o código apresentado na Figura 57. Este arquivo é composto somente pela assinatura das funções (cabeçalhos ou headers) e as variáveis definidas para as estruturas.

Figura 57 – Interface da fila no arquivo *Fila.h*

```
1  /*...5 linhas */
6
7  /*...6 linhas */
13
14 #ifndef FILA_H
15 #define FILA_H
16
17 //Def. a variável de controle da Lista
18 typedef struct no listaEncadeada;
19 //Def. a variável de controle da Fila
20 typedef struct fila controleFila;
21
22 //cria uma fila vazia
23 controleFila* cria(void);
24
25 //Insere os dados no final da Fila
26 void insere(controleFila* ptFila, int info);
27
28 //Imprime a Fila do início ao fim
29 void imprime(controleFila* ptFila);
30
31 //Exclui um elemento do início da Fila
32 int exclui(controleFila* ptFila);
33
34 //Libera todos endereços de memória alocados para Fila
35 void libera(controleFila* ptFila);
36 #endif /* FILA_H */
```

Fonte: Autores, 2018.

Após digitar todo o código no arquivo *Fila.h*, da Fgura 57, clique em salvar. Analisando a referida figura, observamos que nas linhas de 18 e 20 são mostradas as declarações das variáveis *listaEncadeada* e *controleFila*, sendo estas do tipo estrutura nó e fila. A função cria, na linha 23, gera um novo nó dinamicamente, e inicializa os seus

campos, retornando um ponteiro do tipo *contreleFila*. Na linha 26 a função insere recebe como parâmetros o *ptFila* e a *info*. O *ptFila* é um ponteiro definido para a estrutura fila, cuja função é controlar o inicio e o final da lista. Já a *info* é o valor que será armazenado. Na linha 29, temos a função *imprime*, que percorre toda a fila seguindo a ordem do início para o final. Ela recebe como parâmetro o ponteiro *ptFila*. A função *exclui*, na linha 32, tem o objetivo de remover um elemento que esteja no início da fila. Para tal, ele recebe como parâmetro o ponteiro *ptFila*. A função *libera*, na linha 35, remove todos os blocos de memória alocados para a Fila, recebendo como parâmetro o ponteiro de controle da fila, o *ptFila*.

3.3.1.2 Implementação das operações

A seguir será detalha a construção das estruturas e funções. Para construí-las você deve digitar o código da Figura 58 no arquivo *Fila.cpp*. Conforme apresentado na

Figura 58 - Função cria no arquivo *Fila.cpp*

```
1  /*...5 linhas */
6
7  /*...6 linhas */
13
14 #include "Fila.h"
15 #include <stdio.h>
16 #include <stdlib.h>
17
18 //Estrutura paa criar uma lista encadeada
19 struct no{
20     int info;
21     struct no* prox;
22 };
23 //Estrutura para controlar a fila
24 struct fila{
25     listaEncadeada* inicio;
26     listaEncadeada* fim;
27 };
28
29 //Cria uma lista vazia
30 controleFila* cria(void){
31     controleFila* ptFila = (controleFila*)malloc(sizeof(controleFila));
32     ptFila->inicio = NULL;
33     ptFila->fim=NULL;
34     return ptFila;
35 }
36
```

Fonte: Autores, 2018.

Figura 58, além de incluir os arquivos rotineiros, temos que incluir o arquivo da interface criado para fila, o *Fila.h*, conforme exposto na linha 3. Nas linhas de 19 a 22 é criada a estrutura de lista encadeada. Essa estrutura tem a responsabilidade de armazenar as informações e os endereços do próximo nó. Nas linhas 24 a 27 é criada a estrutura para controlar a disciplina de acesso das filas. Logo ela deve conter dois ponteiros para armazenar endereços inicial e final da *fila*. A função *cria*, na linha 30 a 35, aloca memória para o controle da fila, inicializando os ponteiros *(ptFila->inicio e ptFila->fim)* com *NULL*.

Conforme apresentado na Figura 59, a função *insere*, na linha 38, aloca memória para um novo nó da lista encadeada, inserindo a informação no campo *novoNo->info*, na linha 40, e aponta o *novoNo->prox* para NULL, na linha 41. Se a fila for vazia, significa que o elemento a ser inserido é o primeiro, logo temos que

posicionar o ponteiro *ptFila->inicio*, na linha 45, e o *ptFila->fim*, na linha 47, para o endereço do *novoNo*. Caso a lista não seja vazia, ela terá pelo menos um elemento. Sendo assim, é necessário reposicionar somente o ponteiro para o fim, já que os elementos devem ser inseridos somente no final da fila, logo, na linha 43, *ptFila->fim->prox* recebe o endereço do *novoNo*.

A função *imprime*, varre toda a fila na direção do início para o fim, para isso, na linha 52, é declarado o *ptAux*. A este ponteiro, na linha 53, é atribuído o endereço do primeiro elemento da fila, por meio da atribuição *ptAux=ptFila->inicio*, também é estabelecido o critério de parada, o laço irá se repetir até que *ptAux* seja diferente de NULL e, por fim, atualiza-se o endereço para o próximo elemento da lista, *ptAux = ptAux->prox*.

Figura 59 - Função insere e imprime no arquivo *Fila.cpp*

```
37  //Insere os dados no final da Fila
38  void insere(controleFila* ptFila, int info){
39      listaEncadeada* novoNo= (listaEncadeada*)malloc(sizeof(listaEncadeada));
40      novoNo->info = info;
41      novoNo->prox = NULL;
42      if(ptFila->fim != NULL){//Verifica se a lista não é vazia
43          ptFila->fim->prox = novoNo;
44      }else{
45          ptFila->inicio=novoNo;
46      }
47      ptFila->fim = novoNo;
48  }
49
50  //Imprime a Fila do início ao fim
51  void imprime(controleFila* ptFila){
52      listaEncadeada* ptAux;
53      for(ptAux=ptFila->inicio;ptAux != NULL; ptAux=ptAux->prox){
54          printf("\n info->%d",ptAux->info);
55      }
56  }
57
```

Fonte: Autores, 2018.

A função *exclui*, apresentada na Figura 60, inicia declarando o ponteiro *ptAux*, na linha 60, cujo objetivo é auxiliar na exclusão dos elementos, e na linha 61, uma variável do tipo inteiro denominada *info* para armazenar o valor do elemento a ser excluído. Ao final da função o valor excluído é retornado, na linha 74, para a função *chamadora*.

Antes de excluir algum elemento na fila, temos que averiguar se ela não está vazia. Esse teste é realizado na linha 62. Caso ela esteja, o programa é finalizado, na linha 64. Se a fila possui mais de um elemento então *ptAux* é posicionado para o início da fila, na linha 66, e o *ptFila->inicio* é posicionado para o próximo elemento, conforme linha 69, deixando livre o nó atual para remoção. Na linha 73, por meio do comando *free(ptAux)* o elemento atual é removido. Caso a lista contenha somente um elemento, o *ptFila->inicio*, que aponta para o *ptAux->prox*, na linha 69, será NULL. Logo, temos que atualizar o endereço do fim da fila. Sendo assim, o ponteiro *ptFila->Fim*, irá apontar para NULL, apresentado na linha 71, indicando que a fila está vazia.

Figura 60 - Função exclui no arquivo Fila.cpp

```
58    //Exclui um elemento do início da Fila
59    int exclui(controleFila* ptFila){
60        listaEncadeada* ptAux;
61        int info;
62        if(ptFila->fim == NULL){//Verifica se a lista não está vazia
63            printf("Fila vazia, insira elementos!!");
64            exit(1);
65        }
66        ptAux=ptFila->inicio;
67        info=ptAux->info;
68
69        ptFila->inicio = ptAux->prox;
70        if(ptFila->inicio == NULL){//Se a fila ficou vazia
71            ptFila->fim = NULL;
72        }
73        free(ptAux);
74        return(info);
75    }
76
```

Fonte: Autores, 2018.

E, por fim, temos a função *libera*, conforme apresentado na Figura 61. Esta função utiliza dois ponteiros auxiliares, *ptAuxPer* e *PtAuxEnd*. O *ptAuxPer*, na linha 79, percorre toda a fila, liberando os blocos de memória alocado, de acordo com a linha 85, por meio da função *free(ptAuxPer)*. Já o ponteiro *ptAuxEnd*,na linha 80, tem a função de guardar o endereço do próximo bloco de memória a ser removido, na linha 84, logo abaixo, na linha 86, ele atualiza o endereço de *ptAuxPer*.

Figura 61 - Função libera no arquivo *fila.cpp*

```
77    //Libera todos os endereços de memória aolcados para a fila
78    void libera(controleFila* ptFila){
79        listaEncadeada* ptAuxPer;//Ponteiro para percorrer a fila
80        listaEncadeada* ptAuxEnd;//Ponteiro para armazenar o endereço do próximo nó
81        ptAuxPer = ptFila->inicio;
82
83        while(ptAuxPer != NULL){
84            ptAuxEnd = ptAuxPer->prox;
85            free(ptAuxPer);
86            ptAuxPer=ptAuxEnd;
87        }
88        free(ptFila);
89        printf("\n Fila liberada!!!");
90    }
```

Fonte: Autores, 2018.

3.3.1.3 Exemplo de utilização do TAD

Na Figura 62, é apresentado um exemplo de utilização do TAD para a estrutura fila. Vamos programá-lo, para isso, digite o código no arquivo *main.c* e salve com nome *principalFila.cpp*

Figura 62 – Função main no arquivo main.cpp

```
1  /*...5 linhas */
6  /*...6 linhas */
12 #include <cstdlib>
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include "Fila.h"
16
17 using namespace std;
18
19 /*...3 linhas */
22 int main(int argc, char** argv) {
23     controleFila* ptFila = cria();
24     int numero, i;
25     printf("Insere 5 numeros na fila \n");
26     for(i=0;i<5;i++){
27         printf("Digite o numero (%d) a ser inserido na fila: ",i);
28         scanf("%d",&numero);
29         insere(ptFila, numero);
30     }
31     printf("\n Imprime toda a fila");
32     imprime(ptFila);
33     printf("\n\n Exclui 3 numeros da fila");
34     printf("\n Exclui o numero (%d) do inicio da fila",exclui(ptFila));
35     printf("\n Exclui o numero (%d) do inicio da fila",exclui(ptFila));
36     printf("\n Exclui o numero (%d) do inicio da fila",exclui(ptFila));
37     printf("\n\n Imprime toda a lista");
38     imprime(ptFila);
39     libera(ptFila);
40     return 0;
41 }
```

Fonte: Autores, 2018.

Para que o programa no arquivo *main.cpp* funcione adequadamente é necessário incluir o arquivo *Fila.h*, conforme mostra a linha 15. Logo na sequência, na linha 23, é criada uma fila vazia. Nas linhas de 26 a 30, são inseridos 5 elementos do tipo inteiro na fila. Na linha 32, é solicitada a impressão de toda a fila e, nas linhas 34 a 36 é realizada a chamada para a função *exclui*, resultando na exclusão dos três primeiros elementos. Para verificar se exclusão funcionou corretamente, na linha 38, é solicitada novamente a impressão e, por fim, na linha 39, é chamada a função *libera*. Ela faz a liberação dos blocos de memória alocados para toda a fila. O resultado da execução do arquivo principalFila.c é apresentado na figura 63.

Figura 62 – Função main no arquivo main.cpp

```
Insere 5 numeros na fila
Digite o numero (0) a ser inserido na fila: 1
Digite o numero (1) a ser inserido na fila: 2
Digite o numero (2) a ser inserido na fila: 3
Digite o numero (3) a ser inserido na fila: 4
Digite o numero (4) a ser inserido na fila: 5

 Imprime toda a fila
 info->1
 info->2
 info->3
 info->4
 info->5

 Exclui 3 numeros da fila
 Exclui o numero (1) do inicio da fila
 Exclui o numero (2) do inicio da fila
 Exclui o numero (3) do inicio da fila

 Imprime toda a lista
 info->4
 info->5
 Fila liberada!!!
EXECUTAR SUCCESSFUL (tempo total:  11s)
```

Fonte: Autores, 2018.

Veja que os elementos excluídos são sempre aqueles que estão na primeira posição da fila, conforme mostra a Figura 63, fazendo valer a disciplina de acesso para filas, pois o primeiro número inserido, foi o número 1, portanto ele foi o primeiro a ser removido. Vamos aprofundar o conhecimento sobre fila assistindo às videoaulas e praticando o código explicado.

INTERATIVIDADE:
Assista a:

Criando e Destruindo uma Fila Dinâmica
https://www.youtube.com/watch?v=4YXnrKJCWrE

Fila Dinâmica: Informações
https://www.youtube.com/watch?time_continue=11&v=aI-FK1n9Sp30

Fila Dinâmica – Inserção e Remoção
https://www.youtube.com/watch?time_continue=1&v=yO-jgEXbKtME

Atividades – Unidade 3

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Crie um programa que implemente um TAD para lista e pilha que faça as seguintes operações:

1 –exibir todos elementos;
2 –inserir um elemento;
3 –remover um elemento,

2) Faça um programa, utilizando TAD, para implementar uma lista, cujo objetivo é armazenar informações dos alunos, tais como a como matricula, nome e média final, de acordo com a estrutura abaixo.

```
struct aluno{
    int matricula;
    char nome[30];
    float mediaFinal;
};
```

Levando em consideração a estrutura apresentada, faça as seguintes operações:

1 –exibir todos elementos;
2 –inserir um elemento;
3 –remover um elemento.

3) Faça um programa, utilizando TAD, para implementar uma Pilha, cujo objetivo é armazenar informações dos alunos, tais como a como matricula, nome e média final, de acordo com a estrutura abaixo.

```
struct aluno{
    int matricula;
    char nome[30];
    float mediaFinal;
};
```

Levando em consideração a estrutura apresentada, faça as seguintes operações:

1 –exibir todos elementos;
2 –inserir um elemento;
3 –remover um elemento.

4

ÁRVORE

INTRODUÇÃO

Na unidade anterior estudamos Pilhas e Filas. Estas estruturas são ditas estruturas de dados sequenciais, pois elas guardam coleções de dados que são inseridos e acessados sequencialmente. Comumente são caracterizadas como estruturas lineares, pois cada dado tem um único sucessor. Em muitas aplicações computacionais, as organizações dos dados se apresentam de forma não linear, havendo desta forma, a necessidade de representar computacionalmente mais de um sucessor para o elemento em questão. Para resolver tal problema, foram criadas as estruturas de árvores, que são estruturas de dados, não lineares, que permitem implementar algoritmos de forma mais rápida quando comparadas às estruturas lineares, por exemplo, as pilhas e filas, além de oferecer uma representação natural para os problemas que necessitam de uma relação hierárquica entre os seus dados. Sendo assim, ela é usada em diversas aplicações, tais como sistemas de arquivos, interface gráfica, banco de dados, páginas da Internet, dentre outros.

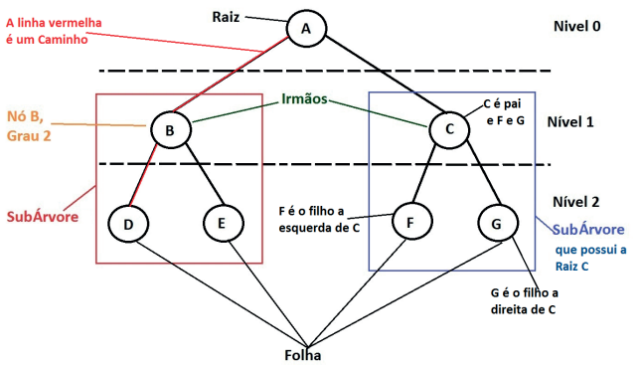
A estrutura de uma árvore nos remete a uma terminologia intuitiva que se baseia em árvores de família, ou árvore genealógica, com os termos pai, filho, ancestral, descendentes, que são palavras comuns utilizadas na definição das relações entre os dados contidos na estrutura. Sendo assim, uma árvore é um tipo abstrato de dados que armazena as entidades do mundo real, tais como pessoas, alunos, professores, peças de motos, etc.

Existem diferentes tipos de árvores. Nesta unidade será abordada uma forma específica de árvore, que é a árvore binária. Este tipo de árvore tem uma característica em particular que é a quantidade de filhos por nós, cada nó pode ter no máximo dois filhos. A seguir será detalhada a terminologia para referida estrutura.

4.1
TERMINOLOGIA

Na estrutura do tipo árvore existem várias terminologias que são utilizadas para descrever as suas funcionalidades, conforme apresentado na Figura 64.

Figura 64 - Terminologias de uma árvore



Fonte: Autores, 2018.

A seguir são descritas as terminologias mais utilizadas (CORMEN, 2002):

- **Raiz:** Nó localizado no topo da árvore que possui sempre nível zero. A raiz não possui um nó pai, nem ancestrais;
- **Arestas:** Também conhecida como ramos, eles representam as ligações entre os nós;
- **Caminho**: É a sequência resultante de nós que foram visitados para completar um percurso de um nó inicial ao nó destino.
- **Nó:** Célula onde se armazenam as entidades (dados), tais como pessoas, peças de carro, na árvore.
- **Grau do nó:** número de subárvores ou número de filhos de um determinado nó.
- **Grau de uma árvore:** Maior grau encontrado entre os seus nós.
- **Nível:** É a soma dos ramos que une a raiz a um nó, ou quantidade de gerações que o nó está da raiz;
- **Altura da árvore:** É o nível mais alto de uma árvore;
- **Pai:** É o nó antecessor, qualquer nó em uma árvore, com exceção da raiz, tem exatamente uma aresta que conduz até outro nó acima dele, que é o pai.
- **Filho:** É o nó descendente de um nó pai, ou seja, qualquer nó pode ter um ou mais arestas descendo para outros nós, e estes denominamos de filhos, e os nós que originaram as arestas são chamados de pai;
- **Folha:** Também definido como nó externo, é o último nó de uma árvore, sendo assim, não possui filhos e tem grau zero;
- **Nó interno:** Nó que possui filhos, e não são nós do tipo folhas;
- **Irmão:** Nós que são filhos de um mesmo pai.
- **Subárvore:** É um subconjunto que possui todos os descentes de um nó.

4.2
ÁRVORES BINÁRIAS

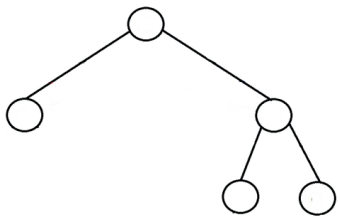
Árvores binárias são estruturas de dados do tipo árvore, em que cada nó interno tem no máximo 2 (dois) filhos (filho esquerdo e filho direito) (PREISS, 2000). Sendo assim, elas são constituídas por um nó raiz e duas subárvores binárias, a subárvore esquerda (sae) e a subárvore direita (sad) e, por sua vez, cada uma destas subárvores ou são nulas ou constituídas por um nó raiz e duas outras subárvores binárias, que são ditas novamente como sendo subárvore esquerda e subárvore direita. Desta forma, cada nó pode ter no máximo duas subárvores, e o grau de cada nó pode ser 0, 1 ou 2.

A seguir é apresentada a classificação das árvores binárias quanto à disposição dos seus nós.

4.2.1 Árvore Estritamente Binária

Uma árvore estritamente binária, conforme representado na Figura 65, é aquela em que cada nó tem 0 ou 2 subárvores, ou seja, nenhum nó tem grau igual a 1 ou não possui um único filho.

Figura 65 -Representação de árvore estritamente binária

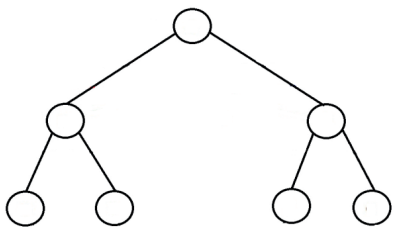


Fonte: Autores, 2018.

4.2.2 Árvore Binária Cheia

Uma árvore é dita como sendo binária cheia, se todos os nós, exceto os nós do último nível, que são as folhas, tiverem exatamente duas subárvores, o que corresponde a ter grau igual a 2, de acordo com a representação, gráfica da Figura 66.

Figura 66 -Representação de árvore binária cheia

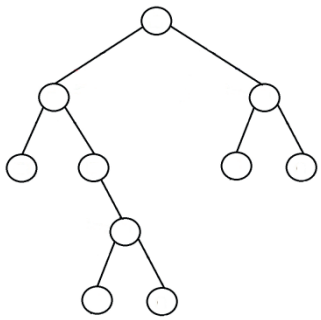


Fonte: Autores, 2018.

4.2.3 Árvore Binária Completa

Uma árvore binária completa, representada na Figura 67, é uma árvore estritamente binária, ou seja, os graus de seus nós podem ser 0 ou 2, na qual seus nós folhas podem estar apenas no último e no penúltimo níveis.

Figura 67 - Representação de árvore binária completa

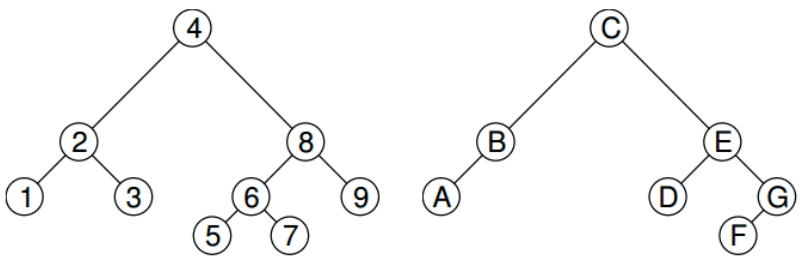


Fonte: Autores, 2018.

4.2.4 Árvore Binária de Busca (ou Árvore Binária Ordenada)

Uma árvore binária de busca é aquela em que todo nó tem uma chave maior que a chave dos seus descendentes à esquerda, e menor que a chave dos seus descendentes à direita. Consequentemente, uma árvore binária de pesquisa só admite uma ocorrência de cada chave, conforme representação da Figura 68.

Figura 68 - Representação de árvore binária de busca



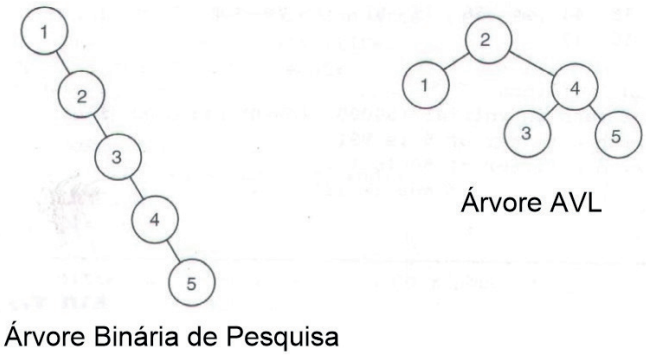
Fonte: Autores, 2018.

4.2.5 Árvore Binária Balanceada (ou Árvore AVL)

Uma árvore binária é considerada balanceada quando, para cada nó, ela possui as alturas de suas subárvores esquerda e direita diferente de no máximo uma unidade. Essa diferença é chamada como fator de balanceamento do nodo. Cada nodo de uma árvore balanceada pode ter fator de balanceamento entre -1 e 1. Idealmente, uma árvore binária é perfeitamente balanceada quando todos os seus nodos têm fatores de balanceamento nulos, como acontece nas árvores binárias cheias. A denominação AVL é uma homenagem aos dois matemáticos russos Adelson-Velskii e Landis que, em 1962, sugeriram este conceito e propuseram algoritmos para manter o balanceamento de uma árvore binária. A altura de uma árvore AVL é no

máximo 45% maior que a altura de uma árvore binária perfeitamente balanceada, considerando o mesmo número de nodos, como mostra a Figura 69.

Figura 69 - Representação de árvore binária balanceada



Fonte: Autores, 2018.

Na subunidade 4.4 iremos implementar uma árvore binária de busca.

4.3
TIPOS DE PERCURSOS EM ÁRVORE BINÁRIA

O percurso em uma árvore é o caminho realizado entre os nós da árvore com o objetivo de consultar ou alterar a informação (o que corresponde a uma visita) neles contidas. Cada nó é visitado uma única vez. Esta visita gera uma sequência linear de nós, cuja ordem depende de como a árvore foi percorrida. Para as árvores binárias, as três formas ou ordens de percurso mais importantes são:

- pré-ordem,
- em-ordem e
- pós-ordem.

A diferença básica entre os 3 (três) percursos é a ordem em que a raiz é visitada, sendo que a subárvore à esquerda é sempre visitada antes da subárvore à direita. A seguir será detalhado cada um dos percursos.

4.3.1 Em-ordem

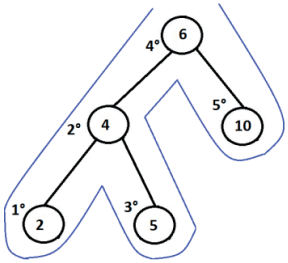
A forma de percorrer os nós da árvore em-ordem corresponde a visitar simetricamente a subárvore esquerda seguida da visita ao nó raiz (visitar um nó significa escrever o seu conteúdo ou executar qualquer outra operação com esse nó), seguida da visita simétrica à subárvore direita.

Assim, os passos para executar as ações supracitadas, podem ser descritos recursivamente em:

1.se árvore vazia, fim
2.percorrer em em-ordem a subárvore esquerda
3.visitar o nó raiz
4.percorrer em-ordem a subárvore direita

A Figura 70 mostra o percurso em-ordem numa arvore binária, como resultado tem-se: 2 4 5 6 10.

Figura 70 - Percurso em-ordem



Fonte: Autores, 2018.

O código que implementa o percurso em-ordem, recursivamente, está apresentado na Figura 71.

Figura 71 – Código do percurso em-ordem

```
54 void arvore_imprime(Arvore* a) {
55     if (!arvore_vazia(a)) {
56         arvore_imprime(a->filhoEsq);
57         printf("[%d] ",a->valor);
58         arvore_imprime(a->filhoDir);
59     }
60 }
```

Fonte: Autores, 2018.

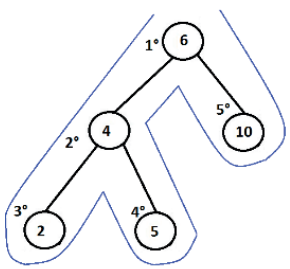
4.3.2 Pré-Ordem

A forma de percorrer os nós da árvore em pré-ordem, corresponde a visitar a raiz, seguidas das visitas em pré-ordem da subárvore esquerda e, depois, da subárvore direita. Assim, os passos para executar as ações supracitadas, podem ser descritos recursivamente em:

1. se árvore vazia, fim
2-visita a raiz.
3-percorre a subárvore da esquerda, em pré-ordem.
4-percorre a subárvore da direita, em pré-ordem.

A Figura 72 mostra o percurso em pré-ordem em uma arvore binária, como resultado tem-se: 6, 4, 2, 5, 10.

Figura 72 - Percurso pré-ordem



Fonte: Autores, 2018.

O código que implementa o percurso pré-ordem, recursivamente, está apresentado na Figura 73.

Figura 73 – Código do percurso em pré-ordem

```
54 void arvore_imprime(Arvore* a) {
55     if (!arvore_vazia(a)) {
56         printf("[%d] ",a->valor);
57         arvore_imprime(a->filhoEsq);
58         arvore_imprime(a->filhoDir);
59     }
60 }
```

Fonte: Autores, 2018.

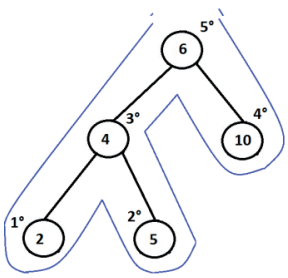
4.3.3 Pós-Ordem

A forma de percorrer, os nós da árvore, em pós-ordem corresponde a visitar o nó, no fim, isto é, depois de ter feito a visita em pós-ordem à subárvore esquerda a visita em pós-ordem à subárvore direita. Assim, os passos para executar as ações supracitadas, podem ser descritos recursivamente em:

1. se árvore vazia, fim
2. percorrer em Pós-Ordem a subárvore esquerda
3. percorrer em Pós-Ordem a subárvore direita
4. visitar o nó raiz

A Figura 74 mostra o percurso em pós-ordem em uma arvore binária, como resultado tem-se: 2, 5, 4, 10, 6.

Figura 74 – Percurso pós-ordem



Fonte: Autores, 2018.

O código que implementa o percurso pós-ordem, recursivamente, está apresentado na Figura 75.

Figura 75 – Código em pós-ordem

```
54 void arvore_imprime(Arvore* a) {
55     if (!arvore_vazia(a)) {
56         arvore_imprime(a->filhoEsq);
57         arvore_imprime(a->filhoDir);
58         printf("[%d] ",a->valor);
59     }
60 }
```

Fonte: Autores, 2018.

4.4
ÁRVORES BINÁRIAS DE BUSCA

Uma Árvore Binária de busca ou Árvore Binária de pesquisa é uma árvore binária em que todos os nós internos contêm algum tipo de informação e, para cada nó, as seguintes propriedades são satisfeitas:

- Todas as informações da subárvore esquerda são menores que a chave da raiz.
- Todas as informações da subárvore direita são maiores que a chave raiz.
- As subárvores à direita e a esquerda são também Árvores Binárias de Busca.

A árvore binária de busca possui, basicamente, as seguintes funções:

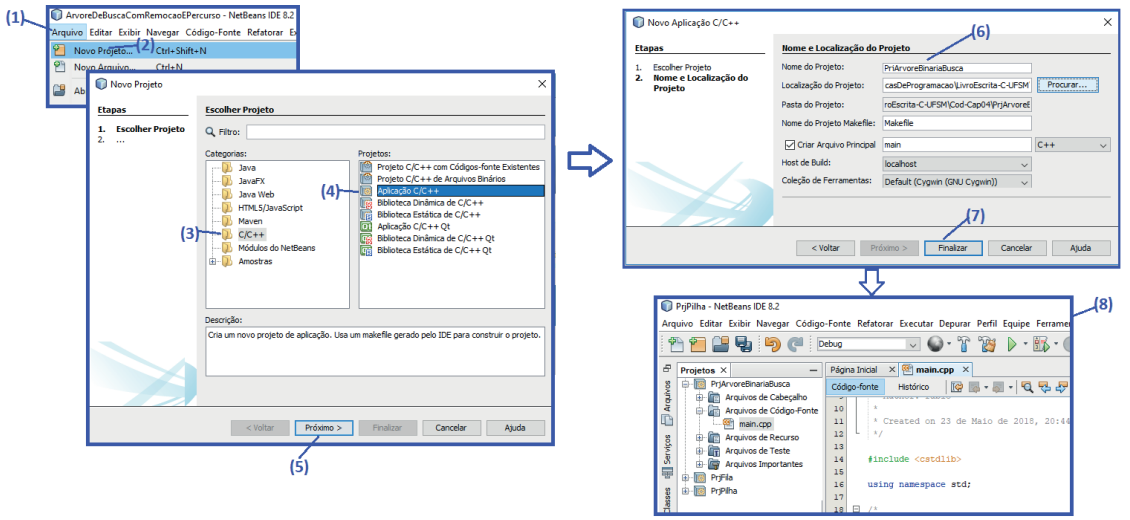
- **Busca:** Localiza um nó com uma determinada chave.
- **Inserção:** Insere um novo nó na arvore, obedecendo aos critérios de inserção para árvores binária de busca.
- **Remoção:** Elimina um nó com uma determinada chave.
- **Travessia:** Uma travessia é uma maneira de percorrer e imprimir as informações em um nó de uma árvore. Há três maneiras simples de percorrer: pré-ordem, em-ordem e pós-ordem.

As funções descritas acima compõem um Tipo Abstrato de Dados que são chamados **dicionários**. A árvore de busca binária implementa dicionários eficientemente, como também outras operações mais complicadas. A seguir são apresentadas as partes do código implementado em Linguagem C.

4.4.1 TAD para árvore binária de busca

Após a explanação teórica supracitada, vamos implementar o TAD para árvore binária de busca no *NetBeans*. Inicialmente crie um novo projeto chamado *PrjArvoreBinariaBusca*. Para isso, faça a sequência de passos de 1 a 8, conforme mostra a Figura 76. Ao finalizar a execução dos passos veja que o *NetBeans* cria um arquivo com o nome *main.c*.

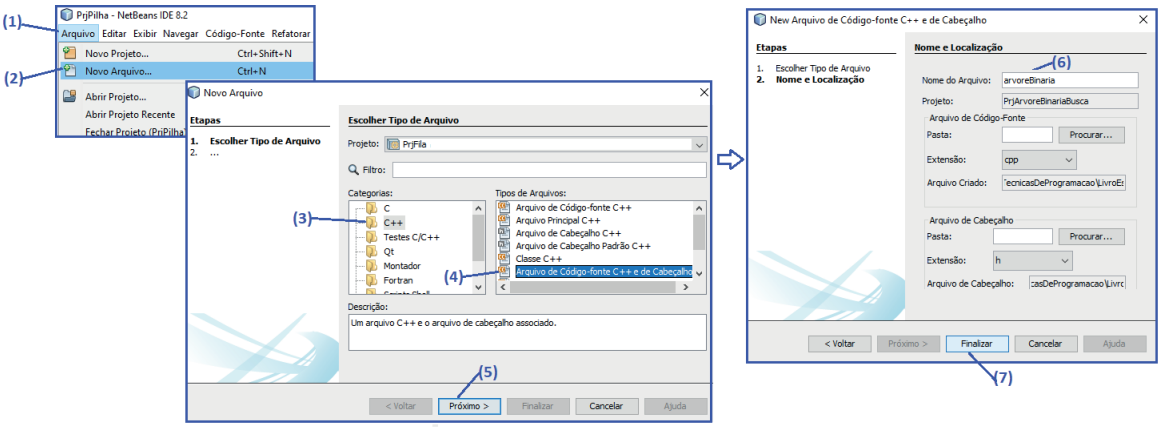
Figura 76 – Criação do projeto *PrjArvoreBinariaBusca*



Fonte: Autores, 2018.

Agora vamos inserir mais dois arquivos, conforme representado na Figura 77.

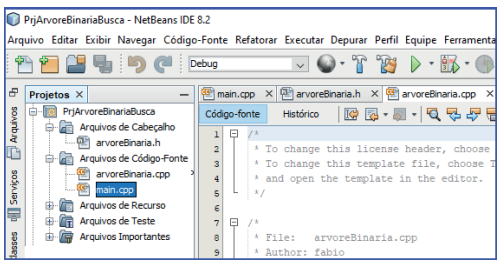
Figura 77 – Criação de novos arquivos no projeto



Fonte: Autores, 2018.

Ao final da execução das ações enumeradas na Figura 77, teremos 3 arquivos no projeto, representados graficamente, na Figura 78. Seguido com a construção do projeto, a próxima etapa é a construção da interface, depois as operações e, por fim, a função principal que fará uso de todas as operações implementadas.

Figura 78 –Novos arquivos criados



Fonte: Autores, 2018.

4.4.1.1 Interface

Para implementar a interface selecione o arquivo *arvoreBinaria.h*. Após ter clicado sobre o nome dele, digite o código apresentado na Figura 79 e salve o arquivo. Repare que o arquivo possui somente a assinatura das funções (cabeçalhos ou *headers*) e as variáveis definidas para as estruturas. Portanto, ele tem o objetivo de mostrar a forma correta das chamadas e a existência das operações.

Figura 79 – Interface da árvore no arquivo *arvoreBinaria.h*

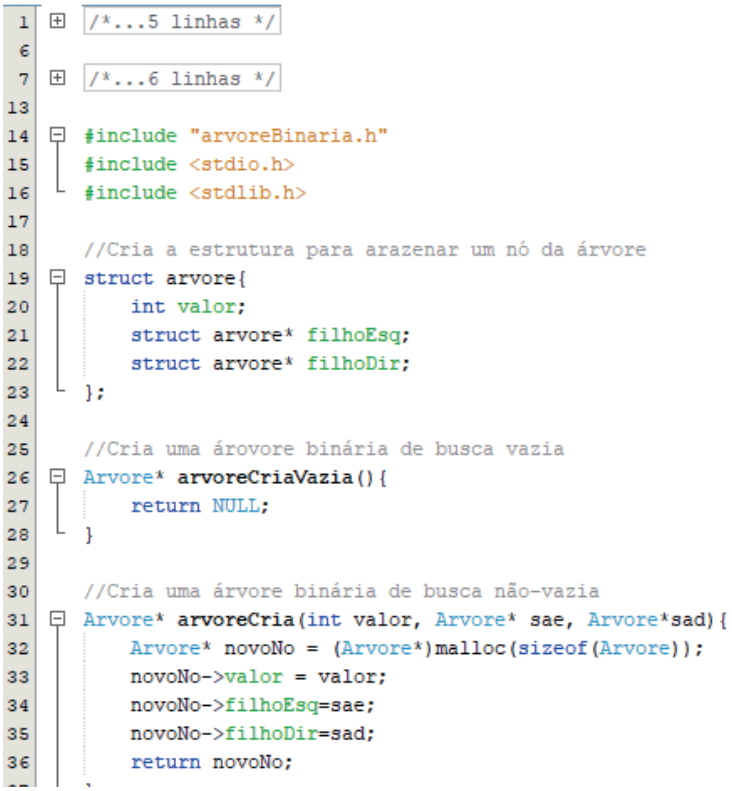
```
1  /*...5 linhas */
6
7  /*...6 linhas */
13
14 #ifndef ARVOREBINARIA_H
15 #define ARVOREBINARIA_H
16
17 //Variável para a estrutura árvore binária de busca
18 typedef struct arvore Arvore;
19
20 //Cria uma árvore binária de busca vazia
21 Arvore* arvoreCriaVazia();
22
23 //Cria uma árvore binária de busca com uma raiz cuja informação é o valor,
24 //e com as com as sub-árvores a esquerda e a direita contendo os
25 // endereçoes  repassados nas variáveis sae e sad, respectivamente.
26 Arvore* arvoreCria(int valor, Arvore* sae, Arvore* sad);
27
28 //Insere um valor inteiro que NÃO exista na árvore. Caso o valor
29 //já exista ele não dever ser inserido.
30 //Ao final, retorna o endereço da raiz da árvore
31 Arvore* arvoreInsere(Arvore* a, int valor);
32
33 //Verifica se o valor passado como parâmetro pertence a árvore a
34 //Ao final, retorna 1 se o valor pertence e 0 caso não perença
35 int arvorePertence(Arvore* a, int valor);
36
37 //Imprime o conteúdo da árvore de forma ordenada
38 void arvoreImprime(Arvore* a);
39
40 //Retorna 1 se a arvore estiver vazia, senão retorna 0
41 int arvoreVazia(Arvore* a);
42
43 //Libera a estrutura da árvore
44 Arvore*arvoreLibera(Arvore* a);
45
46 //Retorna a quantidade de números pares armazenados na árvore
47 int arvoreNumerosPares(Arvore* a);
48
49
```

Fonte: Autores, 2018.

4.4.1.2 Operações

As operações consistem em construir as estruturas e as funções, definidas na interface. Para construí-las você deve digitar o código da Figura 80 no arquivo *arvoreBinaria.cpp*.

Figura 80 – Função *arvoreCria* no arquivo *arvoreBinaria.cpp*



```
1  /*...5 linhas */
6
7  /*...6 linhas */
13
14 #include "arvoreBinaria.h"
15 #include <stdio.h>
16 #include <stdlib.h>
17
18 //Cria a estrutura para arazenar um nó da árvore
19 struct arvore{
20     int valor;
21     struct arvore* filhoEsq;
22     struct arvore* filhoDir;
23 };
24
25 //Cria uma árovore binária de busca vazia
26 Arvore* arvoreCriaVazia(){
27     return NULL;
28 }
29
30 //Cria uma árvore binária de busca não-vazia
31 Arvore* arvoreCria(int valor, Arvore* sae, Arvore*sad){
32     Arvore* novoNo = (Arvore*)malloc(sizeof(Arvore));
33     novoNo->valor = valor;
34     novoNo->filhoEsq=sae;
35     novoNo->filhoDir=sad;
36     return novoNo;
```

Fonte: Autores, 2018.

Analisando a Figura 80 temos, na linha 14, a inclusão da interface que foi criada anteriormente, e nas linhas 15 e 16 as inclusões dos arquivos de bibliotecas considerados como sendo padrão da linguagem de programação C. Na sequência foi criada uma estrutura para a árvore, nas linhas 19 a 23, que tem a responsabilidade de armazenar os valores e os endereços dos filhos à direita e à esquerda do nó corrente.

A função *arvoreCriaVazia*, nas linhas 13 a 15 cria uma arvore binária vazia, retornando para a função chamadora o *NULL*. Por fim, vamos analisar a função *arvoreCria*. Ela recebe um parâmetro do tipo inteiro, um ponteiro para a subárvore à esquerda e outro ponteiro para a subárvore à direita. Na linha 32 ela aloca espaço na memória para um novo nó, do tipo *Arvore*, e nas linhas 33 a 35 faz as devidas atribuições aos seus campos, retornando na linha 36 o endereço do novo nó criado.

A maioria das funções construídas para árvores funcionam de forma recursiva. As duas funções apresentadas na Figura 81 são implementadas usando esta técnica. A função *arvorePertence* recebe como parâmetro um ponteiro que aponta para a raiz da árvore e o valor a pesquisado. Caso este valor já tenha sido inserido na árvore a função retorna 1, na linha 45, caso contrário retorna 0, na linha 43. Estes códigos contidos nestas linhas funcionam como critério de parada da chamada recursiva. Se o elemento que estamos buscando é menor que o valor contido no nó corrente, realiza-se uma chamada recursiva *arvorePertence(a->filhoEsq,valor)* na linha 47, direcionando a busca para o filho a esquerda. Caso contrário realiza-se a chamada recursiva *arvorePertence(a->filhoDir,valor)* que direciona a busca para o filho a direita, de acordo com a linha 49.

A função *arvoreInsere*, assim como a função anterior, recebe dois parâmetros: um ponteiro que aponta para a raiz da árvore e o outro, um valor a ser inserido. Inicialmente, faz-se a verificação se a árvore está vazia, na linha 57, e caso esteja, o ponteiro *a*, que é a raiz da árvore recebe o endereço da árvore criada, conforme a linha 58. Após a inserção do primeiro elemento na árvore, essa função tem que buscar o local adequado para inserir o próximo elemento, levando em consideração que todo nó tem a chave maior que a chave dos seus descendentes à esquerda e menor que a chave dos seus descendentes à direita. Desta forma, quando o valor a ser inserido é menor que o valor do nó corrente é realizada a chamada recursiva *arvoreInsere(a->filhoEsq, valor)* direcionando o caminho para o filho a esquerda, conforme a linha 60. Caso o valor a ser inserido seja maior que o valor do nó corrente, a função recursiva *arvoreInsere(a->filhoDir, valor)* é chamada direcionando o percurso para o filho a direita, conforme a linha 62.

Figura 81 - Funções *arvorePertence* e *arvoreInsere* no arquivo *arvoreBinaria.cpp*

```
38
39  //Verifica se o valor passado como parâmetro pertence a árvorea
40  //e retorna 1 se o valor pertence, senão retorna 0
41  int arvorePertence(Arvore* a, int valor){
42      if(arvoreVazia(a)){
43          return 0;
44      }else if(a->valor == valor){
45          return 1;
46      }else if(valor < a->valor){
47          return arvorePertence(a->filhoEsq,valor);
48      }else{
49          return arvorePertence(a->filhoDir,valor);
50      }
51  }
52
53  //Insere um valor inteiro que NÃO exista na árvore e
54  //retorna o endereço da raiz da árvore. Caso o valor
55  //já exista, retorna o endereço da raiz
56  Arvore* arvoreInsere(Arvore* a, int valor){
57      if(arvoreVazia(a)){
58          a=arvoreCria(valor, arvoreCriaVazia(), arvoreCriaVazia());
59      }else if(valor < a->valor){
60          a->filhoEsq = arvoreInsere(a->filhoEsq,valor);
61      }else if(valor > a->valor){
62          a->filhoDir = arvoreInsere(a->filhoDir,valor);
63      }
64      return a;
65  }
66
```

Fonte: Autores, 2018.

Conforme estudamos, em árvores binárias, podemos utilizar três tipos diferentes de percursos. A função *arvoreImprime* constrói, de forma recursiva, o percurso em ordem, apresentado na Figura 82. A chamada recursiva irá ocorrer tanto direcionada para o lado esquerdo quanto para o direito, até que a árvore esteja vazia, conforme linha 70. A função que testa se a árvore é vazia ou não é construída nas linhas 78 a 80. Ela simplesmente retorna 1 se a árvore for vazia e o caso exista pelo menos um elemento.

Como estamos implementando uma árvore usando alocação dinâmica, antes de finalizarmos o programa temos que liberar todos os blocos de memória utilizados. A função *arvoreLibera* é responsável por esta atividade. Para tal, ela percorre toda a árvore empregando o percurso pós-ordem, nas linhas 85 a 86 e, na linha 87, executa a liberação do espaço de memória alocado.

Figura 82 - Funções *arvoreImprime, arvoreVazia* e *arvoreLibera* no arquivo *arvoreBinaria.cpp*

```
67  //Imprime o conteúdo da árvore de forma ordenada crescente,
68  //usando o percurso em-ordem
69  void arvoreImprime(Arvore* a){
70      if(!arvoreVazia(a)){
71          arvoreImprime(a->filhoEsq);
72          printf("[%d] ",a->valor);
73          arvoreImprime(a->filhoDir);
74      }
75  }
76
77  //Retorna 1 se a árvore estiver vazia, senão retorna 0
78  int arvoreVazia(Arvore* a){
79      return (a == NULL);
80  }
81
82  //Libera a estrutura da árvore binária
83  Arvore* arvoreLibera(Arvore* a){
84      if(!arvoreVazia(a)){
85          arvoreLibera(a->filhoEsq);
86          arvoreLibera(a->filhoDir);
87          free(a);
88      }
89      return NULL;
90  }
91
```

Fonte: Autores, 2018.

Analisando a Figura 83, percebemos que esta função também usa chamadas recursivas, cujo critério de parada, na linha 95, é até que não existam nós a serem percorridos. Neste ponto dizemos que a árvore é vazia. As chamadas recursivas são executadas na linha 98 e, caso o nó corrente seja par, conforme o teste da linha 99, é incrementado o valor de *n*++, na linha 100. Ao final das chamadas recursivas a função retorna o total de números pares, na linha 103.

Figura 83 - Função *arvorePares* no arquivo *arvoreBinaria.cpp*

```
92  //Retorna a quantidade de números pares armazenados na árvore
93  int arvoreNumerosPares(Arvore* a){
94      int n=0;
95      if(arvoreVazia(a)){
96          return 0;
97      }else{
98          n=arvoreNumerosPares(a->filhoEsq) + arvoreNumerosPares(a->filhoDir);
99          if(a->valor %2 == 0){
100             n++;
101         }
102     }
103     return n;
104 }
```

Fonte: Autores, 2018.

4.4.1.3 Exemplo de utilização do TAD

A função main faz as chamadas para as demais funções previamente definidas em *arvoreBinaria.h* e implementadas em *arvoreBinaria.cpp*. Tais chamadas mostram como utilizar o TAD definido anteriormente para a árvore binária de busca. Assim, digite o código apresentado na Figura 84 no arquivo *main.c* e salve-o.

Figura 84 –Função main no arquivo *main.cpp*

```
1  /*...5 linhas */
6  /*...6 linhas */
12 #include <cstdlib>
13 #include <stdlib.h>
14 #include <stdio.h>
15 #include "arvoreBinaria.h"
16
17 using namespace std;
18
19 /*...3 linhas */
22 int main(int argc, char** argv) {
23     Arvore* ptRaiz = NULL;
24     ptRaiz = arvoreCria(6,NULL,NULL);
25     ptRaiz = arvoreInsere(ptRaiz,21);
26     ptRaiz = arvoreInsere(ptRaiz,11);
27     ptRaiz = arvoreInsere(ptRaiz,1);
28     ptRaiz = arvoreInsere(ptRaiz,13);
29     ptRaiz = arvoreInsere(ptRaiz,3);
30     ptRaiz = arvoreInsere(ptRaiz,9);
31     ptRaiz = arvoreInsere(ptRaiz,7);
32
33     printf("Os valores ordenados são: ");
34     arvoreImprime(ptRaiz);
35     printf("\n\n");
36     printf("O numero de valores pares são: %d \n",arvoreNumerosPares(ptRaiz));
37     arvoreLibera(ptRaiz);
38     return 0;
39 }
```

Fonte: Autores, 2018.

Para que o programa apresentado na Figura 84 funcione corretamente, é necessário a inclusão do arquivo de interface *arvoreBinaria.h*, conforme mostra a linha 15. Logo na sequência, na linha 23, é criado um ponteiro para uma árvore e atribuído *NULL* a ele. Na linha 24, é solicitada a criação de uma árvore com o elemento 6. Nas linhas de 25 a 31 é chamada a função *arvoreInsere*, para inserir um elemento em cada linha. Após inserir tais números na árvore, realiza-se uma chamada para a função *imprime*, na linha 34. E ao final, é solicitada a liberação dos blocos de memória alocados para a construção da árvore, conforme mostra na linha 37. O resultado da execução do arquivo *man.cpp* é apresentado na Figura 85.

Figura 85 - Resultado da execução do arquivo *main.cpp*

```
Saída - PrjArvoreBinariaBusca (Executar) ×
Os valores ordenados são: [1] [3] [6] [7] [9] [11] [13] [21]

O numero de valores pares são: 1

EXECUTAR SUCCESSFUL (tempo total:  85ms)
```

Fonte: Autores, 2018.

Ao entender o código acima, você pode ampliar os seus conhecimentos sobre árvores assistindo às videoaulas

INTERATIVIDADE:
Assista a: Árvore Binária: Definição
https://www.youtube.com/watch?v=9WxCeWX9qDs&feature=youtu.be

Árvore Binária: Implementação
https://www.youtube.com/watch?v=TR8ZLUKmcPc&feature=youtu.be

Árvore Binária: Criando e destruindo uma árvore binária
https://www.youtube.com/watch?v=QAJkoJW8bEc&feature=youtu.be

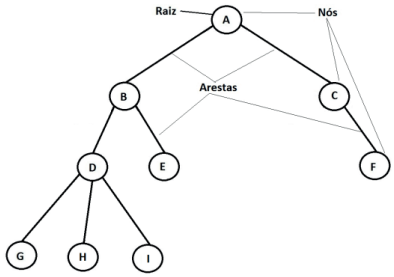
Árvore Binária: informações básicas
https://www.youtube.com/watch?v=qVnNdmx4fOA&feature=youtu.be

Árvore Binária: Percorrendo uma Árvore Binária
https://www.youtube.com/watch?v=z7XwVVYQRAA&feature=youtu.be

Atividades – Unidade 4

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Analise a figura abaixo e responda as seguintes perguntas:



a) Quantas subárvores ela contém?
b) Quais os nós folhas?
c) Qual o grau de cada nó?
d) Qual o grau da árvore?
e) Liste os ancestrais dos nós C e D.
f) Liste os descendentes do nó B.
g) Qual a altura da árvore

2) Após assistir as videoaulas sobre árvore binária, construa o TAD apresentado para árvores binárias de busca e, acrescente na interface uma função para remover um nó com a informação desejada. Sendo assim, uma possível assinatura para esta função é: Arvore* arvoreRemove(Arovre* a, int valor).

5

ALGORITMOS DE ORDENAÇÃO

INTRODUÇÃO

Entende-se por ordenação o ato de colocar os elementos de um conjunto de dados em uma ordem pré-definida, obedecendo a uma ou mais chaves de ordenação. As ordens mais usadas são a numérica e a lexicográfica. (TENENBAUM, 1995).

Em nosso cotidiano existem várias razões para se ordenar um conjunto de dados. Uma delas é a possibilidade de acessar estes dados de maneira mais eficiente, por exemplo, a localização de um número telefônico de uma determinada pessoa em uma agenda. Neste exemplo, a chave é o nome da pessoa e a agenda é a estrutura onde será realizada a pesquisa.

Todas as agendas telefônicas trazem os números telefônicos ordenados pelo nome dos proprietários e, em ordem alfabética crescente, o que facilita o processo de busca. Imagine agora que uma agenda resolve dispor os seus números telefônicos ordenados pela data de inserção do contato. Essa ordem não contribui na busca de um determinado nome, na realidade, ela torna a localização muito difícil, pois os nomes estão, de certa forma, desordenados.

Ao construir um processo de ordenação de dados, temos duas possibilidades. A primeira é quando o conjunto a ser ordenado ainda não foi construído. Neste caso, podemos inserir os elementos na estrutura de forma ordenada. Outra possibilidade é quando o conjunto de dados já foi criado. Nessa opção só nos resta aplicar um dos algoritmos de ordenação. Levando em consideração a localização onde irá ocorrer o processo de ordenação no computador, podemos classificá-lo em ordenação interna e externa. Na interna, as estruturas de dados encontram-se dispostas na memória principal do computador, e na externa as estruturas de dados encontram-se armazenadas na memória secundária, ou no armazenamento auxiliar (no HD *Hard Disk* -disco rígido).

5.1
PRINCIPAIS ALGORITMOS DE ORDENAÇÃO

Normalmente, é difícil encontrar algo em um conjunto de dados quando se utilizam estratégias de busca inadequadas. Primeiramente, antes de realizar a busca por determinado elemento faz-se necessário preparar o conjunto de dados, deixando todos os elementos ordenados. Portanto, a escolha de uma estratégia adequada de busca, passa inicialmente pela ordenação. Logo, temos que conhecer os principais métodos de ordenação.

Na computação existem vários métodos de ordenação, dentre os mais importantes, podemos citar:

- *insertion sort* (ou ordenação por inserção);
- *bubble sort* (ou ordenação por troca),
- *quicksort* (ou ordenação rápida);
- *heap sort* (ou ordenação por *heap*), e o
- *merge sort* (ou ordenação por mistura).

Conforme apresentado anteriormente no exemplo da agenda, a ideia básica da pesquisa envolve, de forma geral, um conjunto de dados e uma chave, sendo que a chave é usada para encontrar um elemento neste conjunto, cujo valor da chave de pesquisa seja o mesmo da chave usada. Seguindo essa premissa, vamos estudar as vantagens e desvantagens dos métodos de:

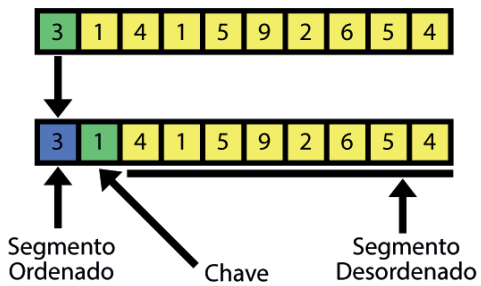
- inserção (*insertion Sort*),
- bolha (*bubble sort*) e o
- *quicksort*.

Nesta unidade todos os exemplos serão aplicados a conjunto de dados, representado por um vetor de inteiros, dispostos na memória principal.

5.2
INSERTION SORT

O algoritmo *Insertion sort*, ou ordenação por inserção, é considerado bem simples de compreender e implementar. Ele apresenta ser eficiente quando aplicado a um número pequeno de elementos, ou seja, quando o vetor a ser ordenado é pequeno. A principal característica deste método consiste em dividir o vetor a ser ordenado em duas partes, sendo que a primeira parte do vetor é considerada como um segmento ordenado. Na sequência temos a chave, cuja função é comparar com os demais elementos ordenados. A outra parte, que está localizada à direita, corresponde ao segmento desordenado (GOODRICH, 2002), (CORMEN et al, 2002), conforme mostra a Figura 86.

Figura 86 - Subdivisão inicial do vetor com o *insertion sort*



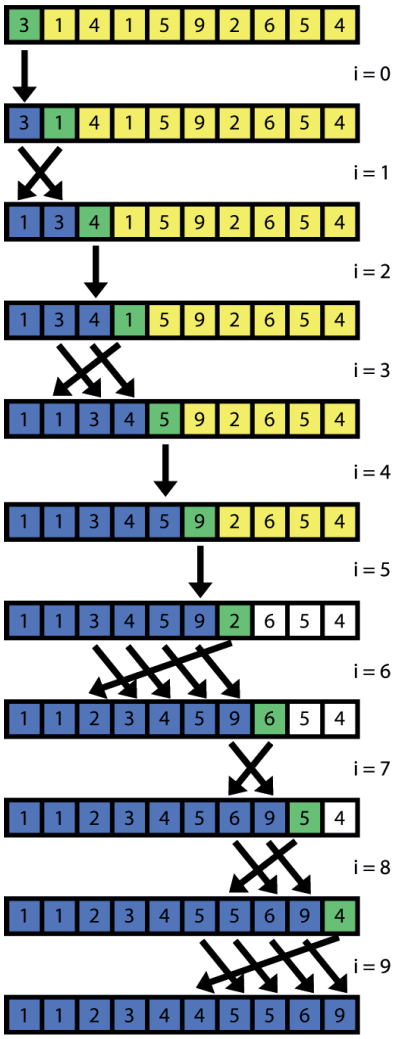
Fonte: Autores, 2018.

Para programar este método temos que primeiro compreender o seu funcionamento. A seguir serão descritos os passos básicos de como ele funciona:

- Considere dois segmentos (sub-vetores) no vetor: ordenado (aumenta) e não ordenado (diminui);
- Ordene por meio da inserção de um elemento por vez, fazendo o deslocamento do elemento do segmento não ordenado para o segmento ordenado;
- Inicialmente, o segmento ordenado contém apenas o primeiro elemento do vetor;
- Realize uma busca sequencial no segmento ordenado para inserir corretamente o elemento do segmento não ordenado;
- Nesta busca, realiza trocas, caso seja necessário, entre elementos adjacentes para ir acertando a posição do elemento que está prestes a ser inserido;
- O processo continua até que todos os elementos estejam ordenados.

A sequência completa de passos para ordenar um vetor é representada graficamente na Figura 87. Para ordenar um vetor com tamanho igual a 10 são necessárias 10 iterações e cada uma delas a chave, elemento na cor verde, é comparado com todos os elementos do segmento ordenado, cor azul. Caso algum deles seja menor que a chave eles devem trocar a posição. Se não for menor, então permanece inalterado.

Figura 86 - Subdivisão inicial do vetor com o *insertion sort*

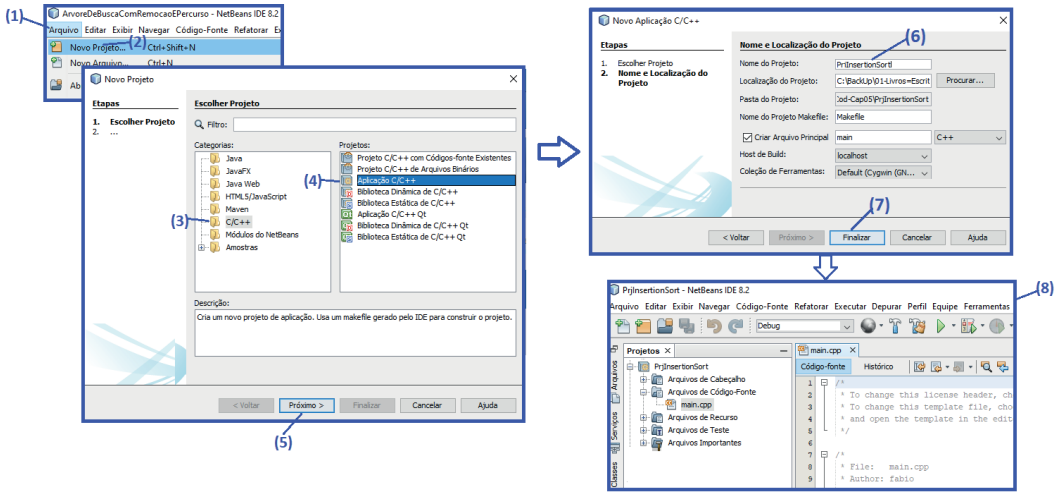


Fonte: Autores, 2018.

5.2.1 Implementação do Insertion sort

Após entendermos o funcionamento do algoritmo, vamos implementar o algoritmo *Insertion sort*, no *NetBeans*. Inicialmente, crie um novo projeto chamado *PrjInsertionSort*, para isso faça a sequência de passos de 1 a 8, conforme mostra a Figura 88. Ao finalizar a execução dos passos, veja que o *NetBeans* cria um arquivo com o nome *main.cpp*.

Figura 88 – Criação do projeto PrjInsertionSort



Fonte: Autores, 2018.

Após a criação do projeto *PrjInsertionSort* vamos implementar as funções, conforme apresentado na Figura 89. Inicialmente é construído o método INSERTIONSORT. Na linha 23 foram declaradas as variáveis *i, j* e *chave*. O i é utilizado para percorrer todo o vetor, veja na linha 25, onde ele é usado no laço *for* como controle, começando em 1, e não em 0, pois o elemento no índice 0 do vetor é considerado como sendo o segmento ordenado. Na linha 26 é atribuído o valor da chave para fazer as devidas comparações. Na linha 27 o j é inicializado como sendo *j=i-1*. Neste caso o *j* tem a função de percorrer todo o segmento ordenado vetor e caso o *vetor[j] > chave*, a chave e o elemento analisado do segmento ordenado são trocados de posição, isso ocorre nas linhas 29 e 31.

Figura 89 - Código da função do *insertion sort*

```
1  /*...5 linhas */
6  /*...6 linhas */
12 #include <cstdlib>
13 #include <stdio.h>
14
15 using namespace std;
16
17 /*...3 linhas */
20
21 //Metodo de inserção direta
22 void InsertionSort(int *vetor, int tamanho){
23     int i,j,chave;
24
25     for(i=1;i<tamanho;i++){
26         chave = vetor[i];
27         j=i-1;
28         while(j>=0 && vetor[j] > chave){
29             vetor[j+1]=vetor[j];
30             j-=1;
31             vetor[j+1]=chave;
32         }
33     }
34 }
35
```

Fonte: Autores, 2018.

A função Imprime, apresentada na Figura 90, linha 37, tem por objetivo imprimir todos os elementos inseridos no vetor, para isso, foi passado como parâmetro o *vetor* e a quantidade de elementos inseridos no vetor, por meio da variável *TamMax*. Para imprimir todos os elementos do vetor o laço de repetição, na linha 39, inicializa o contador i=0 e vai até *i<TamMax*, incrementando o contador *i++*.

A função *main*, na linha 45, faz a chamada para todas as demais funções, passando os devidos parâmetros. Na linha 46 ocorre a declaração do vetor e a inserção de 5 elementos. Na linha 50 é feito o cálculo da quantidade de elementos que foram inseridos no vetor, armazenando em *TamMax*. Antes de chamar *ordenar* o vetor faz-se a impressão do vetor, na ordem em que os elementos foram inseridos, chamando a função *imprime*, na linha 53. Na sequência é solicitada a ordenação chamando a função *InsertionSort*, na linha 56. Por fim, para comprovar se o vetor foi realmente ordenado, chama-se na linha 59 a função *imprime* novamente.

SAIBA MAIS:

Apesar de ser um dos algoritmos de ordenação elementar, o método *insertion sort* comporta-se naturalmente, isto é, trabalha menos quando o vetor já está quase ordenado e chega a sua sobrecarga máximo quando o vetor está ordenado no sentido inverso. Sendo assim ele é uma boa opção para listas que estão quase ordenadas, ou cujo tamanho seja pequena. Logo, a principal desvantagem desse método é quando se trabalha com vetores muito grandes, pois o deslocamento para posicionar um elemento corretamente faz muitas trocas e comparações, deixando o algoritmo lento.

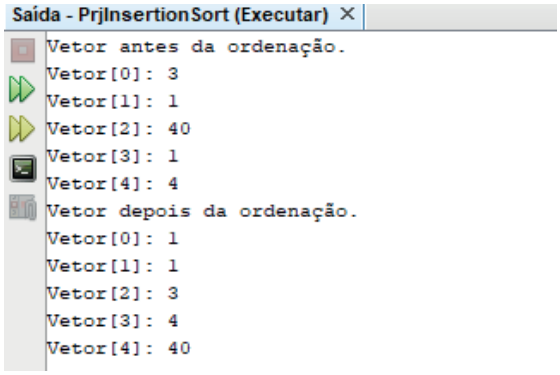
Figura 90 - Código da função do Imprime e main

```
36  //Funcao para imprimir todos os dados o vetor
37  void imprime(int *vetor,int TamMax){
38      int i;
39     for(i=0;i<TamMax;i++){
40          printf("Vetor[%d]: %d \n",i,vetor[i]);
41      }
42  }
43
44  //Funcao principal
45  int main(int argc, char** argv) {
46      int vetor[]={3,1,40,1,4};
47      int i,numBytesDoTipo, numBytesVetor,TamMax;
48      numBytesVetor=sizeof(vetor);//Total bytes ocupados pelo vetor
49      numBytesDoTipo= sizeof(int);//Total bytes ocupados pelo tipo int
50      TamMax = numBytesVetor/numBytesDoTipo;
51
52      printf("Vetor antes da ordenação. \n");
53      imprime(vetor,TamMax)
54              ;
55      //Método de ordenação
56      InsertionSort(vetor,TamMax);
57
58      printf("Vetor depois da ordenação. \n");
59      imprime(vetor,TamMax);
60      return 0;
61  }
```

Fonte: Autores, 2018.

O resultado da execução do programa, apresentado na Figura 90, é apresentado na Figura 91.

Figura 91 – Resultado da ordenação usando o método *InsertionSort*



Fonte: Autores, 2018.

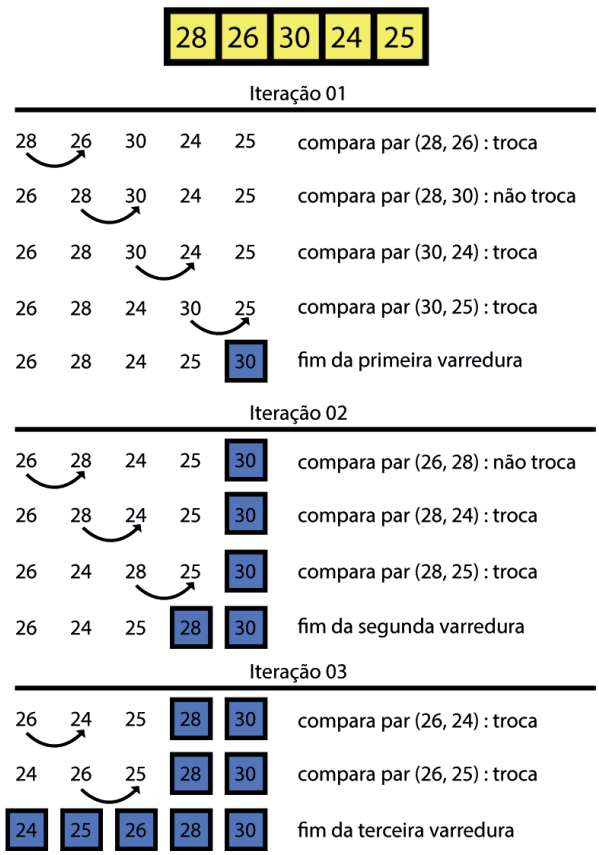
Para que você possa aprender mais sobre o método de ordenação Insertion sort assista à videoaula

INTERATIVIDADE:
Assista a: Ordenação - InsertionSort
https://www.youtube.com/watch?v=79buQYoWszA&feature=youtu.be

5.3
BUBBLE SORT

O algoritmo *bubble sort* ou bolha está entre os mais conhecidos e difundidos métodos de ordenação, mas não é um algoritmo eficiente. Ele ordena por trocas, o que envolve repetidas comparações e, se necessário, troca dois elementos adjacentes no vetor. O nome do algoritmo *bubble sort* se deve ao fato dos elementos contidos no vetor serem comparados a bolhas em um tanque de água, onde cada um procura seu próprio nível, desta forma, os valores mais altos "borbulham" para o final do vetor, para o caso da ordenação crescente (CORMEN et al., 2002), (PREISS, 2000). A ideia principal de funcionamento do algoritmo é apresentada na Figura 92, para isso, suponha que o vetor seja classificado em ordem crescente (LAFORE, 2004).

Figura 92 – Sequência de passos para o *bubble sort*



Fonte: Autores, 2018.

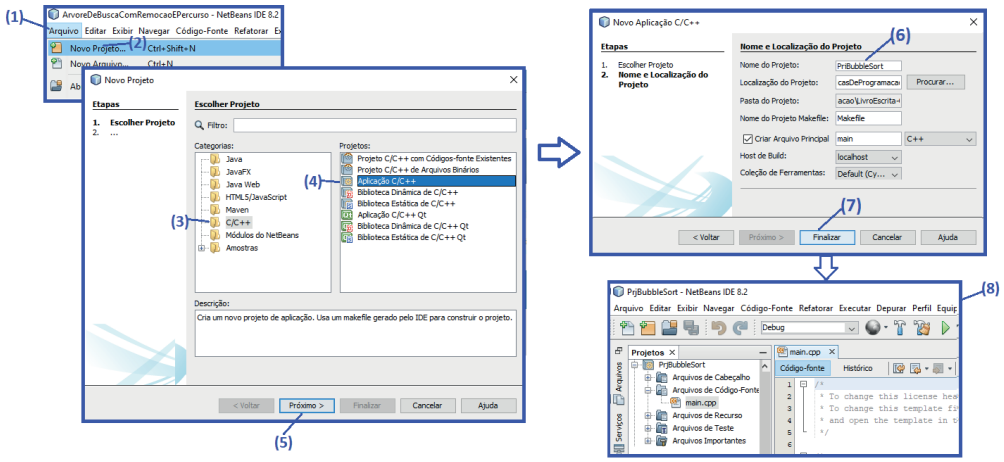
Como podemos observar, o segundo valor do vetor é comparado com o primeiro, conforme mostra a Figura 92. Se o primeiro valor for maior que o segundo, estes têm os índices trocados. Após isso, o terceiro valor é comparado com o segundo. Caso o segundo seja maior que o terceiro, os mesmos terão seus lugares trocados.

Isso ocorrerá para um vetor de tamanho n, n-1 vezes. Ao final da primeira iteração exterior, ou seja, ao finalizar o primeiro percurso de todo o vetor, teremos o maior valor ordenado e posicionado no final do vetor.

5.3.1 Implementação do Bubble Sort

Tendo em vista a explanação teórica sobre o funcionamento do algoritmo, vamos implementá-lo no *NetBeans*. Inicialmente crie um novo projeto chamado *PrjBubbleSort*. Para isso faça a sequência de passos de 1 a 8, conforme mostra a Figura 93. Ao finalizar a execução dos passos, é criado um arquivo com o nome *main.cpp*, onde iremos digitar o código.

Figura 93 – Criação do projeto *PrjBubbleSort*



Fonte: Autores, 2018.

Ao finalizar a criação do projeto *PrjBubbleSort* vamos iniciar a criação da função do método *Bubble Sort*. Observando a codificação do *bubble sort*, na Figura 94, são declaradas as variáveis a, b e temp, na linha 21. A variável *a* é utilizada para percorrer todo o vetor, veja na linha 23, onde ela é usada no laço for como controle. Já a variável *b* é usada no laço for da linha 24 com o objetivo de gerenciar as trocas que ocorrem nas linhas 26 a 28.

Figura 94 - Código da função do *Bubble Sort*

```
1  /*...5 linhas */
6
7  /*...6 linhas */
13
14 #include <cstdlib>
15 #include <stdio.h>
16
17 using namespace std;
18
19 //Metodo de inserção direta
20 void BubbleSort(int *vetor, int tamanho){
21     int a,b,temp;
22
23     for(a=1;a<tamanho;a++){
24         for(b=tamanho-1;b>=a;--b){
25             if(vetor[b-1] > vetor[b]){
26                 temp=vetor[b-1];
27                 vetor[b-1]=vetor[b];
28                 vetor[b]=temp;
29             }
30         }
31     }
32 }
33
```

Fonte: Autores, 2018.

A implementação da função *Imprime*, apresentada na Figura 95, linha 35, tem por objetivo imprimir todos os elementos inseridos no vetor. Para isso utiliza-se o laço de repetição, na linha 37, inicializando o contador i=o e impõe como critério de parada *i<TamMax*, incrementando o contador *i++*.

A função *main*, na linha 43, faz a chamada para todas as demais funções, passando os devidos parâmetros. Vamos analisar os principais pontos: na linha 48 é calculada a quantidade de elementos inseridos no vetor. Antes de chamar a função para ordenar o vetor foi solicitada a impressão, na ordem em que os elementos foram inseridos, chamando a função *imprime*, na linha 51. Na sequência, solicita-se a ordenação do vetor, chamando a função *BubbleSort*, na linha 54. Por fim, para comprovar se o vetor foi realmente ordenado, solicita-se novamente a impressão do vetor, na linha 57.

Figura 95 - Código da função do Imprime e main

```
34  //Funcao para imprimir todos os dados o vetor
35  void imprime(int *vetor,int TamMax){
36      int i;
37     for(i=0;i<TamMax;i++){
38          printf("Vetor[%d]: %d \n",i,vetor[i]);
39      }
40  }
41
42  //Funcao principal
43  int main(int argc, char** argv) {
44      int vetor[]={3,1,4,5,8};
45      int i,numBytesDoTipo, numBytesVetor,TamMax;
46      numBytesVetor=sizeof(vetor);//Total bytes ocupados pelo vetor
47      numBytesDoTipo= sizeof(int);//Total bytes ocupados pelo tipo int
48      TamMax = numBytesVetor/numBytesDoTipo;
49
50      printf("Vetor antes da ordenação. \n");
51      imprime(vetor,TamMax)
52              ;
53      //Método de ordenação
54      BubbleSort(vetor,TamMax);
55
56      printf("\n Vetor depois da ordenação (BubbleSort) \n");
57      imprime(vetor,TamMax);
58      return 0;
59  }
```

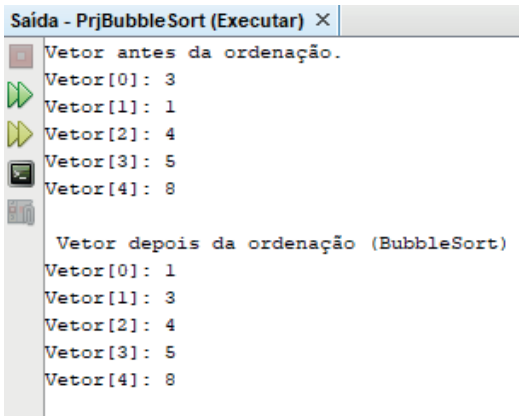
Fonte: Autores, 2018.

O resultado da execução do programa, apresentado na Figura 95, encontra-se na Figura 96.

SAIBA MAIS:

O bubble sort apresenta o mesmo número de comparações, independente do grau de ordenação do vetor. A complexidade desse algoritmo é de ordem quadrática, por isso, ele não é recomendado para programas que precisam de velocidade e operem com quantidade elevada de dados.

Figura 96 – Resultado da ordenação usando o método *Bubble Sort*



Fonte: Autores, 2018.

Busque mais conhecimentos a respeito do método *Bubble Sort* assistindo à videoaula.

INTERATIVIDADE:
Assista a: Ordenação - BubbleSort
https://www.youtube.com/watch?time_continue=2&v=-qU8N_bmebQ4

5.4
QUICK SORT

O *quick sort* faz parte do grupo de algoritmos mais rápido para ordenação interna, que funciona em uma ampla variedade de situações. Este algoritmo é baseado na estratégia de dividir para conquistar. Esta estratégia resolve problemas dividindo-os em dois ou mais subproblemas. Ao final combina as soluções dos problemas menores para obter a solução do problema original (CORMEN et al, 2002), (FARIAS, 2013). Os passos para ordenar um vetor são mostrados na Figura 97 e descritos abaixo.

- Determinar a posição k do elemento pivô para o vetor, usa-se o pivô para dividir o vetor em 2 partes:
  - Esquerda = V[0], ..., V[k-1]
  - Direita = V[k+1], ..., V[n-1]
- Aplicar o algoritmo recursivamente à parte esquerda;
- Aplicar o algoritmo recursivamente à parte direita

O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte, conforme apresenta a Figura 97.

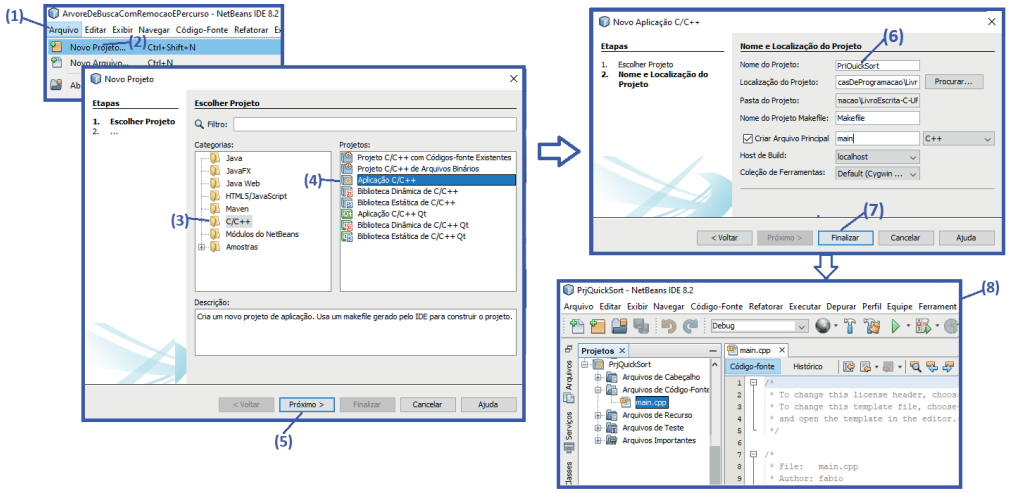
Figura 97 –Sequência de passos do *quick sort*

Fonte: Autores, 2018.

5.4.1 Implementação do Quick Sort

Ao finalizar a parte teórica sobre o funcionamento do *Quick Sort*, vamos implementá-lo no *NetBeans*. Inicialmente crie um novo projeto chamado *PrjQuickSort*, para isso faça a sequência de passos de 1 a 8, conforme mostra a Figura 98.

Figura 98 – Criação do projeto PrjQuickSort



Fonte: Autores, 2018.

A implementação do algoritmo *quick sort* foi dividida em três funções:

- *quickSort()*: Essa função é responsável por fazer as chamadas recursivas;
- *partição()*: Subdivide o vetor original em vetor à esquerda e vetor à direita, por meio dos parâmetros recebidos da função *quickSort()*;
- *troca()*: Faz as trocas para reposicionar os elementos que são considerados fora da ordem.

A seguir vamos analisar cada uma das funções, conforme mostram as Figuras 99, 100 e 101.

Os pontos mais relevantes da função *quickSort*, apresentada na Figura 99, serão detalhados a seguir. A começar na linha 53, que é declarada uma variável do tipo inteiro, sendo esta denominada de pivô. Esta variável serve como ponto de partida na subdivisão do vetor em lado esquerdo, na linha 56, e direito, na linha 57, por meio das chamadas recursivas. Na linha 54, temos a condição de parada da recursividade, ou seja, quando *dir* for menor ou igual à *esq* o vetor não poderá mais ser subdividido.

Figura 99 - Função *quickSort*

```
49
50  //Função quickSort faz as chamadas recursivas
51  //subdividindo o vetor em esquerdo e direito
52  void quickSort(int *vetor, int esq, int dir){
53      int pivo;
54      if(dir > esq){
55          pivo=particao(vetor, esq, dir);
56          quickSort(vetor, esq, pivo-1);
57          quickSort(vetor, pivo+1, dir);
58      }
59  }
```

Fonte: Autores, 2018.

A função partição, na Figura 100, é muito importante neste método de ordenação, pois caso o pivô seja escolhido erroneamente o método pode não funcionar ou perder eficiência. A seguir vamos comentar os aspectos mais importantes no código. Nas linhas de 32 a 34 são declaradas as variáveis a serem usadas na função, dentre elas o x, que recebe o valor de *vetor[dir]*. Na linha 41 ocorre a verificação para chamar a função troca. Se o valor da direita for maior que o da esquerda, então ocorre a chamada da função troca, e a inversão dos valores dentro do vetor é realizada na linha 43. Já na linha 46, é realizada a inserção do valor que foi denominado pivô na posição correta. E por fim, na linha 47, é retornado o índice para calcular a posição do próximo pivô.

Figura 100 - Função partição

```
29
30    //Função de partição do vetor em subvetores a esquerda e direita
31  int particao(int *vetor, int esq, int dir){
32      int x=vetor[dir];
33      int i=esq-1;
34      int j;
35      /*loop de ordenação dos sub vetores, atribuido todos os elementos
36       * menores que o pivo, a posição a esquerda, e os que forem
37       * maiores a direita  */
38      for(j=esq; j<dir;j++){
39          //se o valor da direita for maior que o da esquerda
40          //faz a chamada da função troca
41          if(vetor[j] < x ){
42              i++;
43              troca(&vetor[i],&vetor[j]);
44          }
45      }
46      troca(&vetor[i+1],&vetor[dir]);
47      return i+1;
48  }
```

Figura 101 - Função troca

```
13
14  #include <cstdlib>
15  #include <stdio.h>
16
17  using namespace std;
18
19  //...5 linhas
24  void troca(int *pVetorA, int *pVetorB){
25      int temp=*pVetorA;
26      *pVetorA=*pVetorB;
27      *pVetorB=temp;
28  }
29
```

Fonte: Autores, 2018.

A função troca, apresentada na Figura 101, troca os valores quando solicitado pela função partição. É importante observar que, nesta função, os parâmetros são passados por referência. Também são apresentadas nesta figura, nas linhas 14 e 15 a inclusão das bibliotecas padrões.

Figura 102 - Função *Imprime*

```
60
61    //Funcao para imprimir todos os dados o vetor
62    void imprime(int *vetor,int TamMax){
63        int i;
64        for(i=0;i<TamMax;i++){
65            printf("Vetor[%d]: %d \n",i,vetor[i]);
66        }
67    }
```

Fonte: Autores, 2018.

A implementação da função *Imprime*, apresentada na Figura 102, linha 62, tem por objetivo imprimir todos os elementos inseridos no vetor. Para isso utiliza-se o laço de repetição, na linha 64, inicializando o contador i=0 e impõe como critério de parada *i<TamMax*, incrementando o contador *i++*.

Conforme representado, graficamente, na Figura 102, a função *main* faz as chamadas para todas as demais funções, passando os devidos parâmetros. A seguir vamos analisar os seus principais pontos: na linha 75 é calculada a quantidade de elementos inseridos no vetor. Antes de chamar a função para ordenar o vetor foi solicitada a impressão, na ordem em que os elementos foram inseridos, chamando a função *imprime*, na linha 78. Na sequência é solicitada a ordenação do vetor, chamando a função *quickSort*, na linha 81. Por fim, para comprovar se o vetor foi realmente ordenado, solicita-se novamente a impressão do vetor, na linha 84.

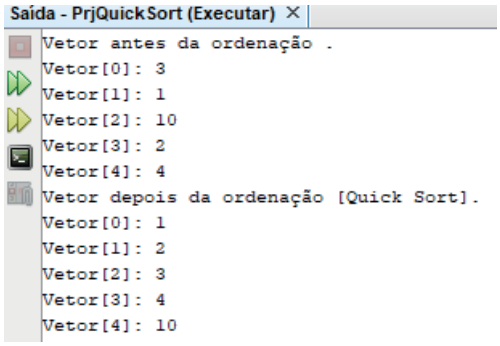
Figura 103 - Função main

```
68
69    //Funcao principal
70    int main(int argc, char** argv) {
71        int vetor[]={3,1,10,2,4};
72        int i,numBytesDoTipo, numBytesVetor,TamMax;
73        numBytesVetor=sizeof(vetor);//Total bytes ocupados pelo vetor
74        numBytesDoTipo= sizeof(int);//Total bytes ocupados pelo tipo int
75        TamMax = numBytesVetor/numBytesDoTipo;
76
77        printf("Vetor antes da ordenação. \n");
78        imprime(vetor,TamMax)
79                ;
80        //Método de ordenação
81        quickSort(vetor,0,TamMax-1);
82
83        printf("Vetor depois da ordenação. \n");
84        imprime(vetor,TamMax);
85        return 0;
86    }
```

Fonte: Autores, 2018.

O resultado da execução do programa, apresentado na Figura 103, encontra-se na Figura 103.

Figura 104 – Resultado da ordenação usando o método *Quick Sort*



Fonte: Autores, 2018.

Após entender os conceitos estudados, referente ao método *Quick Sort*, assista à videoaula para aprofundá-los.

SAIBA MAIS:

É um dos algoritmos mas eficientes que abrange uma ampla gama de aplicações. As diferentes implementações normalmente diferem quanto à escolha do pivô, elemento este que serve como referência para as ordenações, pois este é o elemento principal para obter a máxima performance do algoritmo. Por exemplo, no melhor caso, o tempo de ordenação é de $n \log(n) - n + 1$, sendo necessária a escolha do pivô como sendo o elemento médio do vetor, pois assim, nas seguintes recursões o vetor será dividido exatamente na metade. Já para o pior caso de ordenação, quando o vetor a já se encontra ordenado, o tempo gasto é de $n^2/2$.

INTERATIVIDADE:

Assista a: Ordenação - QuickSort
https://www.youtube.com/watch?v=spywQ2ix_Co

Atividades – Unidade 5

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Implemente o algoritmo Insertion sort, apresentado nas Figuras 89 e 90.

2) Implemente o algoritmo bubble sort, apresentado nas Figuras 93 e 94.

3) Implemente o algoritmo *Quick Sort*, apresentado nas figuras 98 ,99, 100, 101 e 102.

4) Crie um TAD para o *Insertion sort*, que tenhas as especificações mínimas de interface como segue, relembrando a interface deve ser construída no arquivo cuja extensão é *.h:

- void imprimeVetor(int *vetor,int tamanho);
- void insertionSort(int *vetor, int tamanho);
- void BubbleSort(int *vetor, int tamanho);
- void quickSort(int *vet, int esq, int dir);

6

TABELA HASH

INTRODUÇÃO

A tabela de *hash*, também conhecida como espalhamento, é uma estrutura de dados criada para armazenar grandes quantidades de informações, proporcionando rapidez nas operações de inserção e busca, independentemente da quantidade de dados armazenados (CORMEN, 2002). Devido a estas características, ela é largamente utilizada em computação, principalmente para organizar os arquivos em discos nos bancos de dados.

A ideia central, para construir essa estrutura, é a divisão do universo de dados a ser organizado em subconjuntos, sendo estes mais facilmente gerenciáveis, por meio do uso de tabelas.

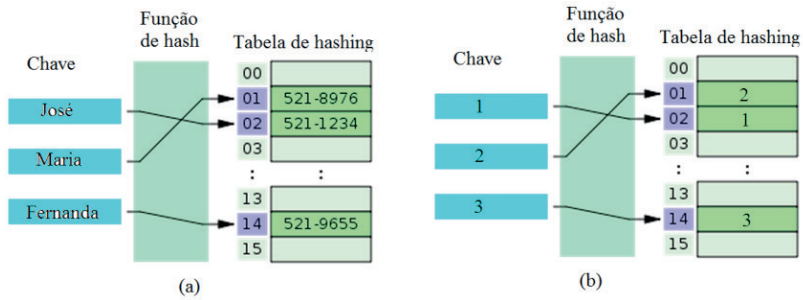
Na apresentação da estrutura de tabelas, a busca por uma chave ocorre sempre por meio de comparações. Uma alternativa de busca em tabelas dá-se por meio do cálculo da posição que uma chave ocupa na tabela através de uma função. Na tabela *hash* os registros são armazenados na tabela e estes são diretamente endereçados a partir da aplicação de uma função sobre a chave. O método para implementar a tabela *hash* é composto por dois conceitos centrais (PREISS, 2000):

- Função de hashing: Calcula o mapeamento entre valores de chaves e as entradas na tabela;
- Tabela de hash: É um arranjo, construído por meio de estrutura de dados, no qual existe um mapeamento entre seus índices e as chaves de um subconjunto de registros, desta forma ela permite o acesso aos registros.

Na Figura 104 temos duas representações, a primeira (a) mostra uma estrutura composta por dois campos: nome e telefone, sendo que a chave é uma *string*. Já na segunda (b), temos uma estrutura com um campo que é um número. Inicialmente quando queremos inserir uma chave, aplica-se a função de *hashing* sobre a chave e a partir deste resultado, decide-se em qual das posições da tabela o elemento deve ser armazenado. Para realizar a uma busca, aplica-se novamente a função de *hashing* sobre a chave e usa-se este resultado para definir a subespaço de procura. Portanto, ao invés de realizar a busca em toda a lista, faz-se uma busca em um subconjunto da lista. Assim, consegue-se uma busca eficiente. Embora os elementos não fiquem dispostos ordenadamente, o *hash* permite uma melhora na eficiência da localização individual dos elementos em relação às listas.

Como vimos, a questão crucial neste método, é a escolha adequada da função de *hashing*, tal função deve garantir uma distribuição uniforme dos dados na tabela, evitando as colisões.

Figura 104 - Tabela *hash*



Fonte: Autores, 2018.

6.1
FUNÇÃO HASH

Como vimos anteriormente, a função *hash*, (função de dispersão ou transformação) é a responsável por gerar um índice a partir de uma determinada chave. Caso ela seja mal escolhida, toda a tabela terá um baixo desempenho. Idealmente esta função *hash* deveria sempre fornecer índices únicos para as chaves de entrada. Desta forma, a função perfeita é aquela que, para quaisquer entradas X e Y, sendo X diferente de Y, ela forneça saídas diferentes. Na prática, funções *hash* perfeitas ou quase perfeitas são encontradas apenas onde a colisão é intolerável, por exemplo, nas funções *hash* da criptografia.

Criar uma função *hash* que satisfaça as condições supracitadas não é tarefa fácil, pois existem muitos métodos tais como: divisão, multiplicação e o *fibonacci*. A seguir vamos estudar o método da divisão.

6.2
MÉTODO DA DIVISÃO

Este método é um dos mais usados para criar uma função *hash*. Ele pode ser usado tanto para chaves numéricas, quanto para cadeia de caracteres. Independentemente do tipo da chave ele só precisa de duas informações, o valor a ser armazenado, que chamamos de chave, e o tamanho da tabela *hashing*, denominado Max. A Figura 105 apresenta a função hash para chaves numéricas do tipo *float*. Veja que o método calcula o resto da divisão da chave por *Max*, e o resultado do cálculo é o índice onde devemos armazenar o elemento.

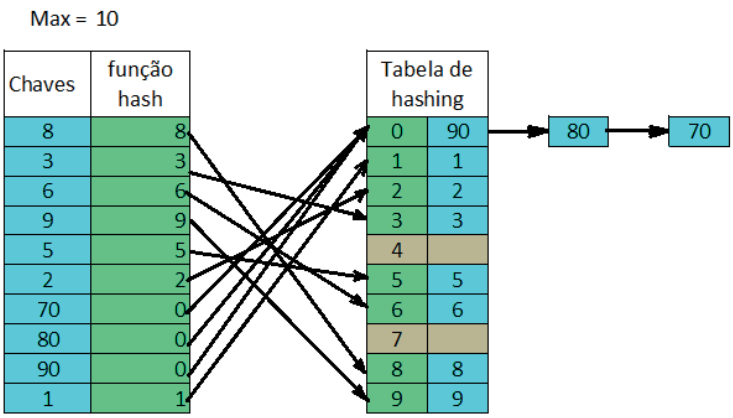
Figura 105 - Método da divisão para chaves do tipo *float*

```
1
2  int funcaoHash(float chave){
3      return (int) chave%Max; //Função Hash
4  }
```

Fonte: Autores, 2018.

O principal problema deste método é que elementos distintos podem receber o mesmo índice. Por exemplo, quando queremos limitar o número máximo de elementos armazenados em 10, e o conjunto de chave de entrada em {8, 3, 6, 9, 5, 2, 70, 80, 90, 1}, teremos problemas de repetição de índices para elementos diferentes. Veja como fica o resultado do cálculo e da inserção das chaves, ambos apresentados na Figura 106.

Figura 106 - Cálculo e inserção de chaves



Fonte: Autores, 2018.

Observe que os números 70, 80 e 90 tiveram os seus índices repetidos, logo eles foram inseridos na mesma posição da tabela. Para minimizar o problema de repetição de índices, para números distintos, neste método é aconselhável usar números primos para definir o tamanho máximo da tabela. Voltando ao nosso exemplo, o tamanho máximo da tabela definido anteriormente é 10 e, examinando a sequência de

números primos, temos (2, 3, 5, 7, 11, 13, 17, 19, 23..,). Logo, chegamos à conclusão que, por proximidade, podemos usar o número 11.

A seguir é apresentada a versão deste método para chaves do tipo cadeia de caracteres, conforme mostra a Figura 107.

Figura 107 - Cálculo e inserção de chaves

```
 8  int funcaoHash(char chave[]){
 9      int valHash = 0, k = 0;
10      while (chave[k] != '\0'){
11          valHash = valHash + int(chave(k));
12          k++;
13      }
14      return (int)(valHash % Max);
15  }
```

Fonte: Autores, 2018.

Ao usar chaves do tipo caractere, o método transforma cada letra em um valor numérico e soma todas elas, como apresentado na linha 11 e, ao final, faz o cálculo do resto da divisão de *valHash* por *Max*, retornando o índice, que será o endereço para armazenar o elemento.

6.3
TRATAMENTO DE COLISÕES

Por mais que se tente encontrar uma função *hash* eficiente, em aplicações práticas é difícil conseguir evitar o problema de colisão de chaves. Definimos colisão como sendo o ato de atribuir um mesmo endereço para dois valores de chave diferentes. Na Figura 106, apresentada anteriormente, temos colisão entre os números 70, 80 e 90. Veja que eles possuem o mesmo endereço ou índice na tabela *hashing*. Sendo assim, o primeiro elemento a chegar fica com a posição. O problema é como iremos tratar o armazenamento dos outros dois. Por causa das colisões, muitos analistas de sistemas aliam as tabelas *hashing* com alguma outra estrutura de dados, na tentativa de redirecionar os elementos para um espaço vago na memória, por exemplo, uma lista encadeada ou até mesmo uma árvore. Assim, existe uma variedade enorme de alternativas para resolver este problema, as mais usuais são:

- Endereçamento aberto;
- Encadeamento separado.

A seguir, será detalhada cada uma dessas alternativas.

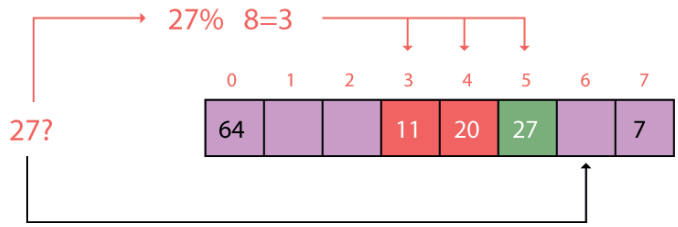
6.3.1 Endereçamento aberto

Nesta solução, todos os elementos estão armazenados na própria tabela *hash*, podendo ser construída somente por meio de vetores. Consequentemente, esta solução dispensa o uso de ponteiros. Basicamente, ao tratar colisões por meio de endereçamento aberto, quando uma nova chave é mapeada para uma posição já ocupada, outra posição é indicada para esta chave, e se esta posição também estiver ocupada, o algoritmo tem que continuar a busca por uma nova posição, por todo o vetor, até que uma posição vazia seja encontrada. O nosso problema atual consiste em definir uma forma adequada para encontrar esse espaço vazio. Contudo existem vários métodos, como a exploração linear, exploração *quadrática* e *hash duplo*, dentre estes vamos analisar o primeiro.

6.3.1.1 Exploração linear

Nesta solução, a busca por uma posição vazia ocorre de forma linear, ou seja, quando a chave deveria ser inserida em uma posição que está ocupada, buscamos novas posições seguintes. Na Figura 108 temos um exemplo onde queremos inserir o número 27. Veja, a primeira posição sugerida pela função *hash* é a 3, mas lá já existe um número armazenado. Então iremos tentar a próxima posição que é a 4, também está ocupada. Então iremos para a posição 5, que também está ocupada e, por fim, chegamos na posição 6 que está vaga, então vamos inserir nela o número 27.

Figura 108 - Exploração linear



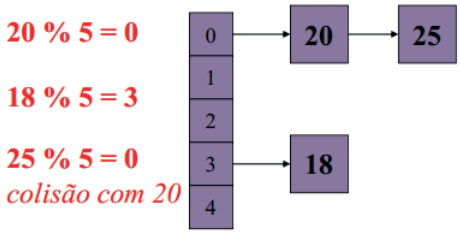
Fonte: Autores, 2018.

Embora a exploração linear seja de fácil implementação, ocasiona, na maioria das vezes, um problema conhecido como agrupamento primário. Esta falha, forma longas sequências de posições ocupadas, aumentando o tempo médio para pesquisar uma posição vazia.

6.3.2 Encadeamento separado

Na solução por endereçamento aberto a solução para as colisões eram resolvidas procurando por uma posição vazia. No encadeamento separado a solução é inserir, em cada índice da tabela *hashing*, uma lista encadeada. Desta forma, os elementos que possuem chaves iguais são inseridos na mesma lista, conforme mostra a Figura 109.

Figura 109 - Encadeamento separado



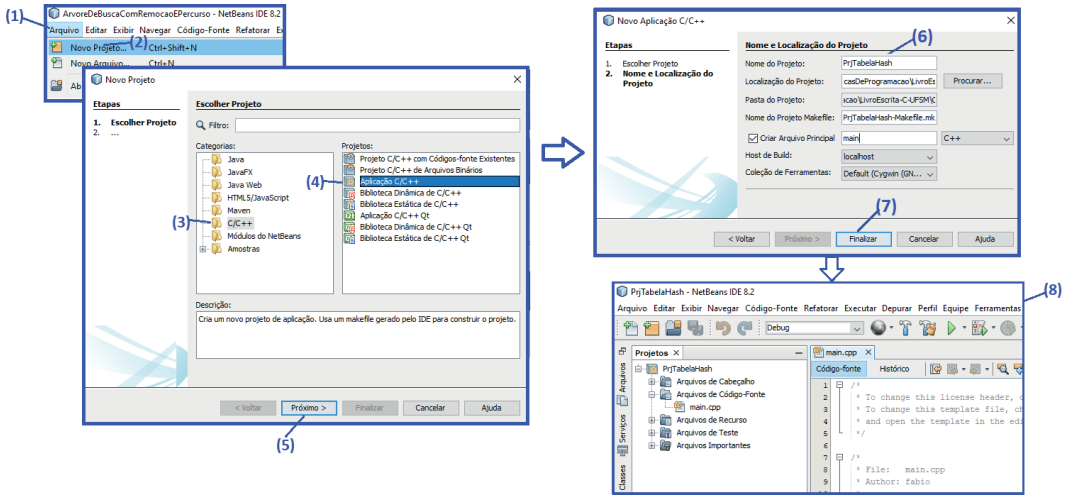
Fonte: Autores, 2018.

Nesta estratégia, ao analisarmos a Figura 109, percebemos que a inserção é rápida, pois estamos usando uma lista encadeada e inserindo no início. Desta forma a inserção sempre vai ocorrer no índice calculado. Já a busca por um elemento, leva tempo proporcional ao tamanho da lista armazenada em cada índice.

6.4 IMPLEMENTAÇÃO

Como já foi apresentada a parte teórica sobre o funcionamento da tabela *Hash*, vamos implementá-la, no *NetBeans*. Inicialmente crie um novo projeto chamado *PrjTabelaHash*. Para isso faça a sequência de passos de 1 a 8, conforme mostra a Figura 110.

Figura 109 - Encadeamento separado



Fonte: Autores, 2018.

Para simplificar a nossa implementação vamos montar uma tabela *hash*, cuja finalidade é a de armazenar chaves numéricas. Para isso, a codificação construída usa o encadeamento separado, por meio de lista encadeada, cuja função *hash* a ser utilizada é o método da divisão. Desta forma, a implementação do algoritmo da Tabela *hashing* foi dividida nas seguintes funções:

- *inicializaTabela()*: Inicializa todos os ponteiros do vetor da tabela;
- *funcaoHash()*: Calcula o índice para inserir uma chave na tabela;
- *imprime()*: Imprime todas as chaves armazenadas na tabela;
- *insere()*: Insere uma chave usando o tratamento de colisão por encadeamento separado usando lista encadeada;
- *exclui()*: Exclui uma chave na tabela;
- *busca()*: Faz a busca de uma chave na tabela.

A seguir, vamos implementar cada uma destas funções. Para isso abra o arquivo *main.c* e implemente o código apresentado nas Figuras 111 e 112.

Figura 111 - Função *inicializaTabela*

```
1  /*...5 linhas */

7  /*...6 linhas */

13
14 #include <cstdlib>
15 #include <stdio.h>
16 #include <stdlib.h>
17 #define Max 10  //Número máximo de chaves a ser armazenado
18
19 using namespace std;
20
21 //Nó da lista encadeada para armazenar
22 //as informações na tabela hashing
23 struct hash{
24     int info;
25     struct hash *prox;
26 };
27 typedef struct hash Hash;
28
29 //Inicializa todos os ponteiros do vetor da tabela hashing
30 void inicializaTabela(Hash *tabela[]){
31     int i=0;
32     for(i=0; i< Max;i++){
33         //Limpa os ponteiros
34         tabela[i]=NULL;
35     }
36 }
37
```

Fonte: Autores, 2018.

A função inicializa, apresentada na Figura 111 atribui NULL, na linha 34, a todos os índices da tabela. A tabela é definida como sendo um vetor de ponteiros do tipo *Hash*, que é a nossa estrutura definida nas linhas 23 a 26. Esta declaração será apresentada com mais detalhes quando estivermos estudando a função *main*. Não se esqueça de digitar todo o código apresentado na Figura111.

Na Figura 112 é apresentada a função *Hash*, que gera o índice para inserir uma chave na tabela por meio do método da divisão.

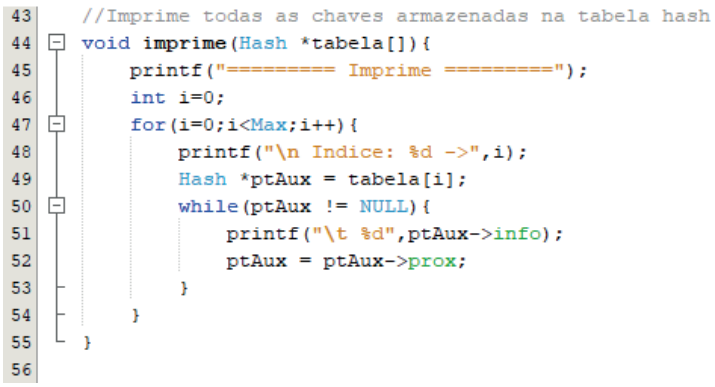
Figura 112 - Função *funcaoHash*

```
38 //Retorna a posição na tabela hash que a chave deve ser inicializada
39 int funcaoHash(int chave){
40     return (int) chave % Max; //função Hash com o método de divisão
41 }
42
```

Fonte: Autores, 2018.

Já a função *imprime* (conforme Figura 113) mostra todas as chaves que foram armazenadas na tabela. Para isso ela tem que percorrer toda a *tabela[i]*, por meio do laço *for* na linha 47. Na sequência, para cada índice da *tabela[i]*, ela percorre o conteúdo armazenado, que é uma lista encadeada, até chegar ao final e imprime o campo *info*, essas ações ocorrem nas linhas 50 a 53.

Figura 113 - Função *imprime*

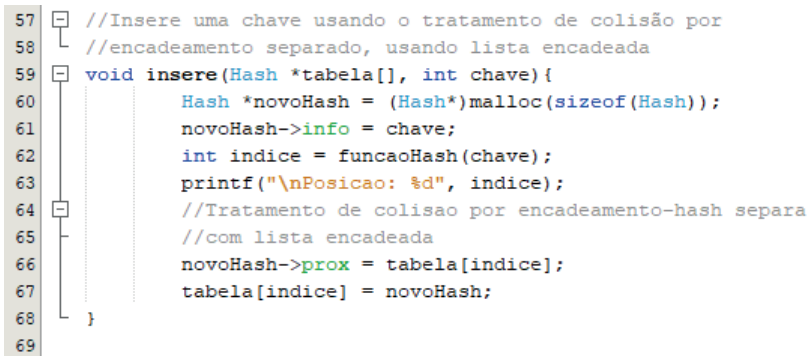


```
43   //Imprime todas as chaves armazenadas na tabela hash
44   void imprime(Hash *tabela[]){
45       printf("========== Imprime =========");
46       int i=0;
47       for(i=0;i<Max;i++){
48           printf("\n Indice: %d ->",i);
49           Hash *ptAux = tabela[i];
50           while(ptAux != NULL){
51               printf("\t %d",ptAux->info);
52               ptAux = ptAux->prox;
53           }
54       }
55   }
56
```

Fonte: Autores, 2018.

Conforme mostra a Figura 114, a função insere, tem por objetivo inserir uma chave na tabela *hash*. Para isso, deve-se alocar espaço de memória para o *novoHash* e inserir nele o campo chave, conforme mostram as linhas 60 e 61. O próximo passo é gerar o índice por meio da função *funcaoHash*, na linha 62. Além de inserir uma chave, esta função deve tratar o problema da colisão, usando o método de encadeamento separado com lista encadeada. Esse processo é executado nas linhas 66 e 67.

Figura 114 - Função *insere*



```
57   //Insere uma chave usando o tratamento de colisão por
58   //encadeamento separado, usando lista encadeada
59   void insere(Hash *tabela[], int chave){
60           Hash *novoHash = (Hash*)malloc(sizeof(Hash));
61           novoHash->info = chave;
62           int indice = funcaoHash(chave);
63           printf("\nPosicao: %d", indice);
64           //Tratamento de colisao por encadeamento-hash separa
65           //com lista encadeada
66           novoHash->prox = tabela[indice];
67           tabela[indice] = novoHash;
68   }
69
```

Fonte: Autores, 2018.

A função exclui (Figura 115) é a nossa função mais complexa a ser implementada. Ela recebe como parâmetros a tabela e a chave. Com estas informações, ela deve calcular o índice de localização da chave dentro da tabela, na linha 72, e na sequência, passa então a procurar pela chave no índice calculado. Esse processo tem 3 possibilidades: a chave a ser excluída não existe, essa verificação é feita na linha 76; a chave encontrada está localizada no início da lista encadeada, conforme mostram as linhas de 78 a 81; e, por fim, a chave a ser deletada encontra-se em uma posição diferente do início, conforme mostram as linhas de 87 a 96. Veja que nesta situação temos 2 ponteiros: *ptAux*, usado para percorrer os nós da lista, e *ptAnt* usando para armazenar o endereço do nó anterior ao apontado pelo *ptAux*. O processo de percorrer todos os nós da lista é executado na linha 89. Nela o ponteiro *ptAux* recebe o endereço do próximo nó a sua frente e esse processo só termina quanto não tiver mais nós ou encontrar a chave. Ao encontrar a chave, na linha 90, ele atualiza o endereço do nó anterior que é *ptAnt->prox*, com o endereço do nó, logo a sua frente, que está

localizado em *ptAux->prox*. Após serem realizadas as atualizações de endereços, o nó indicado por *ptAux* é removido, conforme mostra a linha 92.

Figura 115 - Função *exclui*

```
70  //Exclui uma chave na tabela
71  void exclui(Hash *tabela[], int chave){
72      int indice = funcaoHash(chave);
73      //ponteiros para percorrer a lista e fazer atualização de endereços
74      Hash *ptAux = tabela[indice];
75      //chave não foi inserida
76      if(tabela[indice] != NULL){
77          //chave se encontra no inicio da lista
78          if(tabela[indice]->info == chave){
79              tabela[indice]=tabela[indice]->prox;
80              free(ptAux);
81              printf("\nNumero excluido!");
82          //Chave se encontra nas demais posições
83          }else{
84              Hash *ptAnt = tabela[indice];
85              //Percorre a lista até encontrar a chave ou até
86              //que ela seja vazia
87              while(ptAux->prox != NULL){
88                  ptAnt = ptAux;
89                  ptAux = ptAux->prox;
90                  if(ptAux->info == chave){//Econtrou a chave na lista
91                      ptAnt->prox = ptAux->prox;
92                      free(ptAux);
93                      printf("\nNumero excluido!!");
94                      break;
95                  }
96              }
97              if(ptAux->prox == NULL){//Chegou ao final da lista
98                  printf("\nNumero não encontrado: ");
99              }
100         }
101     }else{
102         printf("\n Numero nao encontrado!!!");
103     }
104 }
105
```

Fonte: Autores, 2018.

A *busca*, apresentada na Figura 116, tem, por objetivo, o de encontrar uma chave que foi inserida na tabela. Para isso, a função recebe como parâmetros a tabela e a chave. Após calcular o índice para a provável localização dentro da tabela, na linha 108, faz-se então, a partir do índice, uma varredura na lista encadeada para averiguar a inserção desta chave. Este processo está implementado nas linhas 111 a 118. E, ao final, a função retorna à localização da chave encontrada, na linha 113 ou uma mensagem informando que a chave não foi encontrada, na linha 120.

Figura 116 - Função *busca*.

```
106   //Busca por uma chave, na tabela hash
107   void busca(Hash *tabela[], int chave){
108       int indice = funcaoHash(chave);
109       Hash *ptAux = tabela[indice];
110
111       while(ptAux != NULL){
112           if(ptAux->info == chave){
113               printf("\n A chave [%d] encontra-se no ", ptAux->info);
114               printf("indice->%d",indice);
115               break;
116           }
117           ptAux = ptAux->prox;
118       }
119       if(ptAux==NULL){
120           printf("\n Numero nao encontrado!!");
121       }
122   }
123
```

Fonte: Autores, 2018.

A função principal ou função *main*, apresentada na Figura 117, é onde são realizadas as chamadas para as outras funções. Incialmente temos a criação do vetor de ponteiros do tipo *Hash*, cujo tamanho máximo foi definido por Max. A seguir é realizada a inicialização da tabela, linha 131 e, na sequência, é apresentado um menu de opções cuja escolha deve ser feita pelo usuário, nas linhas 135 a 142. Dependendo da opção escolhida o programa pode inserir uma chave, na linha 148, excluir uma chave, na linha 154, fazer uma busca, na linha 160, ou imprimir toda a tabela, na linha 163.

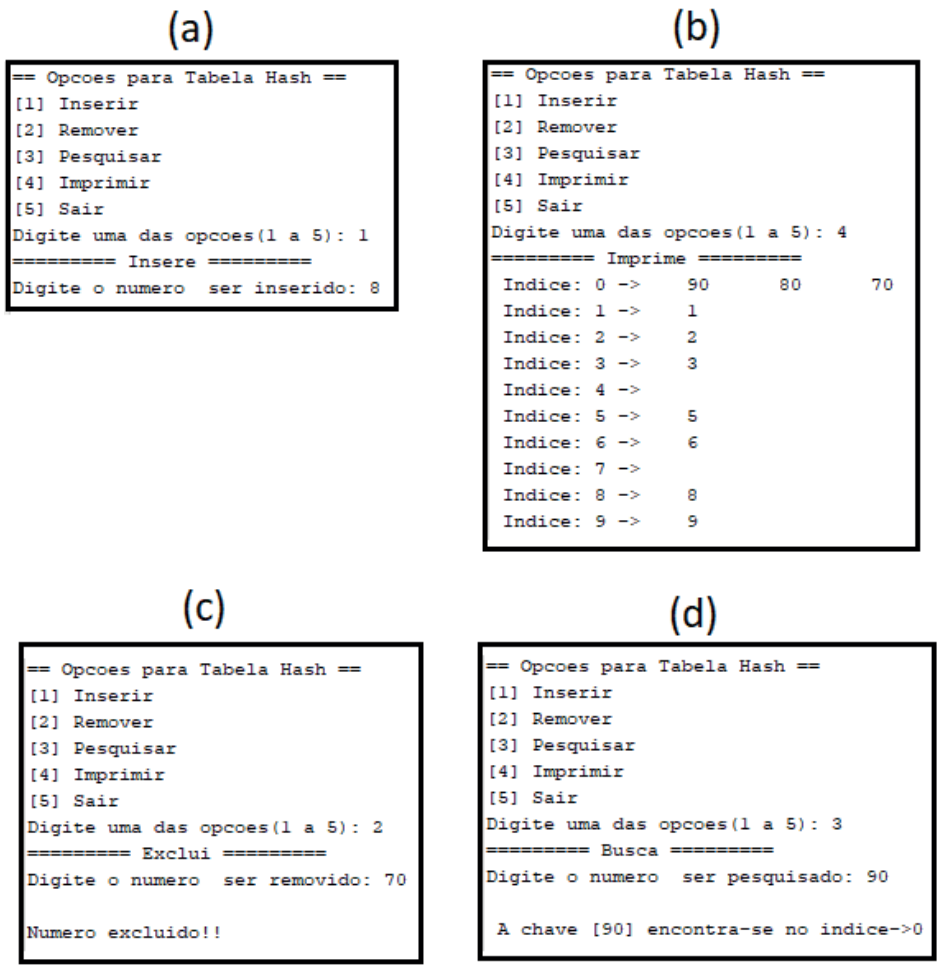
Figura 117 - Função *main*.

```
/*
 * Funcao principal para construir a tabela hashing
 */
int main(int argc, char** argv) {
    int op=0;
    int chave;
    Hash *tabela[Max];//Cria vetor de ponteiros para a tabela hash
    inicializaTabela(tabela);//inicializa a tabela com NULL
    //menu de opcoes
    while(op != 5){
        op=0;
        printf("\n\n== Opcoes para Tabela Hash == \n");
        printf("[1] Inserir \n");
        printf("[2] Remover \n");
        printf("[3] Pesquisar \n");
        printf("[4] Imprimir \n");
        printf("[5] Sair \n");
        printf("Digite uma das opcoes(1 a 5): ");
        scanf("%d",&op);
        switch(op){
            case 1:
                printf("========= Insere =========");
                printf("\nDigite o numero  ser inserido: ");
                scanf("%d",&chave);
                insere(tabela, chave);
                break;
            case 2:
                printf("========= Exclui =========");
                printf("\nDigite o numero  ser removido: ");
                scanf("%d",&chave);
                exclui(tabela, chave);
                break;
            case 3:
                printf("========= Busca =========");
                printf("\nDigite o numero  ser pesquisado: ");
                scanf("%d",&chave);
                busca(tabela, chave);
                break;
            case 4:
                imprime(tabela);
                break;
        }
    }
    return 0;
}

```

Fonte: Autores, 2018.

A Figura 118 apresenta o resultado da execução das 4 ações principais do nosso programa, são elas: a)inserir; b) imprimir; c) remover; d) pesquisar. Analisando a opção remover, pode-se concluir que foram inseridos os elementos {8, 3, 6, 9, 5, 2, 70, 80, 90, 1}, também notamos que os números {90, 80,70} sofreram colisão. Logo esse problema foi tratado pelo método de encadeamento separado com lista encadeada, conforme podemos observar na Figura 118, letra b.

Figura 118 - Resultado da execução do programa *tabelaHash.c*.



Fonte: Autores, 2018.

Atividades – Unidade 6

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Implemente o algoritmo para implementar a tabela hash, cujo código está apresentado nas figuras 111, 112, 113, 114, 115, 116 e 117.

2) Implemente o TAD para a tabela *hash* com todas as funções explanadas acima *(inicializaTabela; funcaoHash(); imprime(); insere();exclui() e busca())* e acrescente uma função para **destruir toda a lista**, ou seja, que libere todos os espaços de memória alocados.

REFERÊNCIAS

CELES, W; CERQUEIRA, R; RANGEL, J. L. **Introdução à estrutura de dados**. Rio de Janeiro: Elsevier, 2004.

CORMEN, C. L; RIVEST, R; STEIN, C. **Algoritmos - Teoria e Prática**. Rio de Janeiro: Campus, 2002.

DEITEL, H. M; DEITEL, P. J; CHOFFNES, D. R. **Sistemas Operacionais**. São Paulo: Pearson Prentice Hall, 2005.

FARIAS, R. **Estrutura de Dados e Algoritmos**. Disponível em: <http://www.cos.ufrj.br/~rfarias/cos121/aula_11.html>. Acesso em: 2 jun. 2018.

GOODRICH, M. T; TAMASSIA, R. **Estrutura de dados e algoritmos em Java**. 2. ed., Porto Alegre: Bookman, 2002.

LAFORE, R. **Estrutura de dados & algoritmos em java**. Rio de Janeiro: Ciência Moderna Ltda., 2004

PREISS, B. R. **Estrutura de dados e algoritmos**. Rio de Janeiro: Elsevier, 2000.

SCHILDT, H. C, **Completo e Total**. São Paulo: Makron, MCGraw-Hill, 1990.

TENENBAUM, A. M; LANGSAM ,Y; AUGENSTEIN, M. **Estrutura de dados usando C**. São Paulo: Pearson Makron Books, 1995.

UNICAMP. **Estrutura de Dados**. Disponível em :<http://www.ft.unicamp.br/liag/siteEd/>.Acesso em: 13/06/2018.

ZIVIANI, N. **Projeto de algoritmos: com implementação em Java e C++**. São Paulo: Thomson, Learning, 2007.

CONSIDERAÇÕES FINAIS

Prezado estudante: toda vez que você for representar, conceitualmente, os dados de um problema do mundo real, no mundo computacional, os conceitos referentes a técnicas de programação serão necessários, já que as informações inseridas no computador consistem em um conjunto cuidadosamente selecionado de dados do objeto real, e tais informações são armazenadas nas estruturas apresentadas neste livro.

Cabe ressaltar que, por meio deste material, você teve uma visão geral de como o computador armazena estruturalmente as informações de um programa, na memória, bem como é realizada a busca e ordenação destas informações.

Inicialmente este material descreveu a estrutura de listas, que se apresentam como base para muitos tipos estruturas de dados. Com as listas, podemos criar programas para resolver diversos problemas práticos, como armazenar as informações de uma lista telefônica ou de uma lista de notas dos alunos.

Também estudamos os conceitos de pilhas, que é um tipo de lista linear, onde todas as inserções, remoções e acessos são realizadas em um único extremo (topo da pilha). Neste sentido, o último elemento que foi armazenado, é o primeiro a ser removido da pilha.

Estudamos, ainda, as filas. Neste tipo de estrutura o primeiro elemento a entrar é o primeiro a sair. Logo, os elementos entram por um lado e saem por outro.

Aprendemos a estrutura de árvore, que são aplicadas quando há a necessidade de representar computacionalmente mais de um sucessor para um elemento. As árvores são estruturas de dados não lineares, que permitem implementar algoritmos de forma mais rápida quando comparadas às estruturas lineares, como as pilhas e filas. Além disso, por ser uma estrutura não linear, uma árvore permite maior agilidade na busca por informações.

Ao armazenar as informações em uma das estruturas de dados estudadas, geralmente, surge a necessidade de ordenar as informações levando em consideração um dado específico, que chamamos de chave. Para tal finalidade estudamos os Algoritmos de Ordenação (*insertion Sort, bubble sort e quicksort*).

E, por fim, estudamos a Tabela *Hash,* que é uma estrutura de dados criada para armazenar grandes quantidades de informações, proporcionando rapidez nas operações de inserção e busca, independentemente da quantidade de dados armazenados.

Reparem que o livro foi construído especialmente para o nosso curso, levando em consideração a distribuição dos conteúdos de acordo com o cronograma da disciplina de Técnicas de Programação. Sendo assim, você, futuro profissional Licenciado em Computação, deve seguir rigorosamente os prazos de entrega das atividades, tendo em vista que este livro é a base para os conteúdos futuros, tais como os que serão estudados na disciplina de "*Linguagem de Programação II*".

APRESENTAÇÃO DOS AUTORES

A definição das unidades deste material foram idealizadas a partir das experiências, em sala de aula, dos docentes do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria) / Campus Frederico Westphalen. Segue um breve resumo do currículo dos autores:

Fábio Parreira: Possui graduação em Ciência da Computação pela UNITRI (Centro Universitário do Triângulo), Especialista em Produção de Material Didático para EaD pela UFAM (Universidade Federal do Amazonas), mestrado em Processamento Digital de Imagens pela UFU (Universidade Federal de Uberlândia) e doutorado em Inteligência Artificial e Informática de Sinais Biomédico pela UFU (Universidade Federal de Uberlândia). Atualmente é Professor Associado do Departamento de Tecnologia da Informação no campus de Frederico Westphalen - RS da UFSM (Universidade Federal de Santa Maria). Suas áreas de interesse envolvem, principalmente: Jogos Educacionais Digitais, Inteligência Artificial e Educação a Distância.

Guilherme Bernardino da Cunha: Possui graduação em Ciência da Computação, mestrado em Ciências com Ênfase em Inteligência Artificial e Processamento Digital de Imagens e Doutorado em Ciências com ênfase em Engenharia Biomédica pela Universidade Federal de Uberlândia. Atualmente é professor Adjunto da UFSM - Universidade Federal de Santa Maria - Campus Frederico Westphalen. Tem experiência na área de Ciência da Computação atuando principalmente nos seguintes temas: Engenharia de Software, Inteligência Artificial, Análise de Séries Temporais, Epidemiologia, Banco de Dados, Redes Neurais Artificiais, Algoritmos Genéticos, Bioinformática.

Teresinha Letícia da Silva: Possui graduação em Informática pela Universidade Regional Integrada do Alto Uruguai e das Missões, Especialização em Ciência da Computação pela Universidade Federal de Santa Catarina e Mestrado em Ciência da Computação pela Universidade Federal de Santa Catarina. Atualmente é Professora Assistente em Regime de Dedicação Exclusiva da Universidade Federal de Santa Maria no campus de Frederico Westphalen. Tem experiência na área de Ciência da Computação, com ênfase em Software Básico, atuando principalmente nos seguintes temas: internet, educação a distância, computação gráfica, realidade virtual e aumentada e recuperação de informações.

Adriana Soares Pereira: Possui graduação em Informática pela Universidade Regional do Noroeste do Estado do Rio Grande do Sul, mestrado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul, doutorado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Membro

do Banco de Avaliadores do SINAES - Ministério da Educação e Professora Adjunta da Universidade Federal de Santa Maria. Tem experiência na área de Ciência da Computação, com ênfase em Sistemas Multiagentes, atuando principalmente nos seguintes temas: ambientes educacionais, desenvolvimento de sistemas, ensino a distância e sistemas especialistas.