



APOSTILA DE

ESTRUTURA DE DADOS

EM JAVA

Sumário

CAPÍTULO 1 - INTRODUÇÃO	3
1.1 ESTRUTURAS DE DADOS	3
1.2 ALOCAÇÃO ESTATICA x DINÂMICA	4
1.3 PONTEIRO ou APONTADOR	5
1.4 TIPOS ESTRUTURADOS DE DADOS HETEROGÊNEOS (REGISTROS).....	6
CAPÍTULO 2 - LISTAS LINEARES	8
CAPÍTULO 3 - LISTA LINEAR SIMPLEMENTE ENCADEADA	11
3.1 ELIMINAÇÃO E INSERÇÃO DE ELEMENTOS NA LISTA.....	12
3.2 LISTA SIMPLEMENTE ENCADEADA COM DESCRITOR	12
3.3 OPERAÇÕES BÁSICAS DA LISTA SIMPLEMENTE ENCADEADA	14
3.3.1. <i>Classe da declaração do nó:</i>	14
3.3.2 <i>Classe da declaração da Lista Simplemente Encadeada:</i>	14
3.3.3. <i>Aplicativo para manipular a lista simplesmente encadeada:</i>	16
CAPÍTULO 4 - LISTA LINEAR POR CONTIGUIDADE	18
4.1 COMPARAÇÃO ENTRE A REPRESENTAÇÃO POR CONTIGUIDADE E POR ENCADEAMENTO	18
4.2 OPERAÇÕES BÁSICAS DA LISTA POR CONTIGUIDADE	19
CAPÍTULO 5 - LISTA LINEAR DUPLAMENTE ENCADEADA	22
5.1 OPERAÇÕES BÁSICAS DA LISTA DUPLAMENTE ENCADEADA	23
CAPÍTULO 6 - LISTAS LINEARES CIRCULARES	27
6.1 SIMPLEMENTE ENCADEADA	27
6.2 DUPLAMENTE ENCADEADA	27
CAPÍTULO 7 - PILHA	28
7.1 OPERAÇÕES BÁSICAS DA PILHA POR CONTIGUIDADE	30
CAPÍTULO 8 - FILAS	33
8.1 FILA CIRCULAR.....	35
8.2 OPERAÇÕES BÁSICAS DA FILA CIRCULAR	35
CAPÍTULO 9 - RECURSIVIDADE	38
CAPÍTULO 10 - LISTA NÃO LINEAR - ÁRVORE	41
10.1 ÁRVORE BINÁRIA	43
10.2 ÁRVORES DE PESQUISA	44
10.3 ÁRVORE BINÁRIA DE PESQUISA	44
10.4 OPERAÇÕES BÁSICAS DE ÁRVORE BINÁRIA DE PESQUISA.....	48
REFERÊNCIAS.....	53

Capítulo 1 - INTRODUÇÃO

Na evolução tecnológica no mundo computacional, um fator de relevante importância é a forma de armazenar as informações. De nada adiantaria o grande desenvolvimento do hardware e do software, se a forma de armazenamento e tratamento da informação não acompanhasse esse desenvolvimento. Por isso a importância das estruturas de dados, que são formas otimizadas de armazenamento e tratamento das informações eletronicamente.

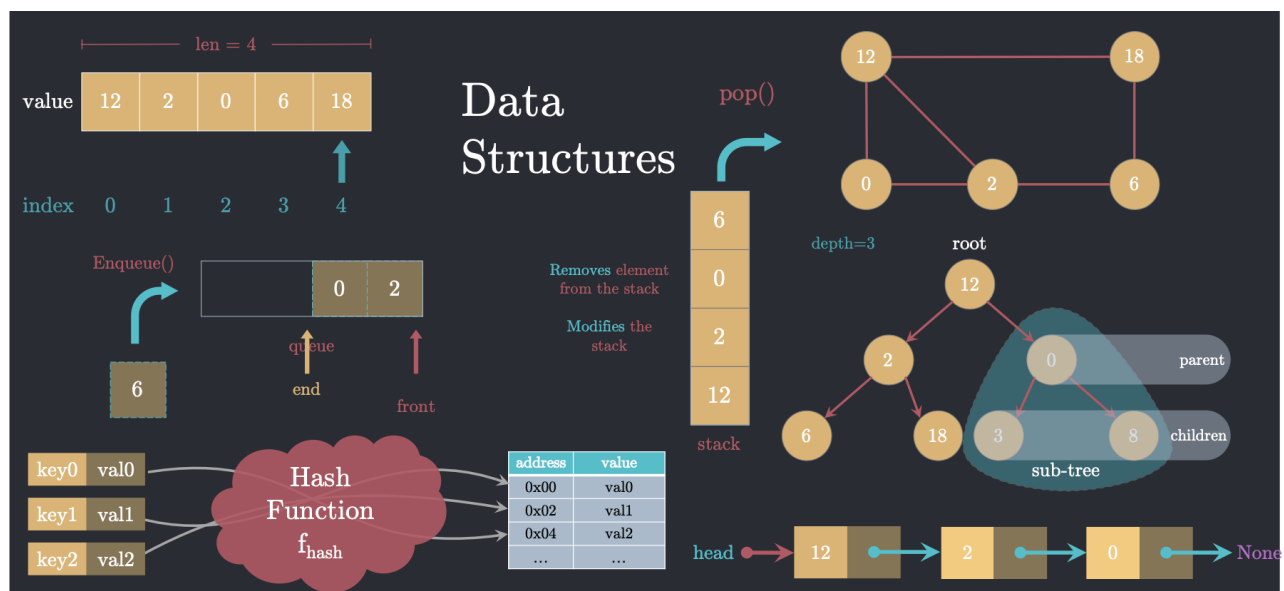
As estruturas de dados, na sua maioria dos casos, foram espelhadas em formas naturais de armazenamento do nosso dia a dia, ou seja, nada mais são que a transformação de uma forma de armazenamento já conhecida e utilizada no nosso mundo para o mundo computacional. Por isso, cada tipo de estrutura de dados possui vantagens e desvantagens e cada uma delas tem sua área de atuação (massa de dados) ótima.

Esta apostila foi elaborada com o objetivo de introduzir o aluno, que já possua um conhecimento da linguagem de programação Java, no mundo das estruturas de dados, bem como mostrar a teoria (conceitos) e a prática (implementação) das estruturas de dados mais utilizadas no dia a dia do profissional de informática.

O aluno até o período acadêmico anterior, somente teve contato com estruturas de dados do tipo vetores e matrizes, e deve estar familiarizado com suas facilidades e limitações de uso. Agora, novas estruturas de dados serão apresentadas, sendo que, todas elas serão estruturas de dados dinâmicas, ou seja, crescem e diminuem de acordo com a necessidade, para isso, serão necessárias várias rotinas que controlam esta dinamicidade das estruturas.

Este capítulo destina-se a apresentar alguns conceitos fundamentais para o entendimento das estruturas de dados.

1.1 ESTRUTURAS DE DADOS



Objetivo: Otimizar o funcionamento do programa através da representação dos dados.

Fatores a considerar: ✓ Tempo de acesso aos dados
 ✓ Ocupação da memória

Resolução de um problema consiste na abstração da realidade, em geral através da definição de um conjunto de dados que representa a situação real, e na escolha da estrutura para representar estes dados.

Escolha da representação dos dados dependente das operações sobre eles.

Segundo Wirth¹,

“Programa = Algoritmo + Estrutura de Dados”

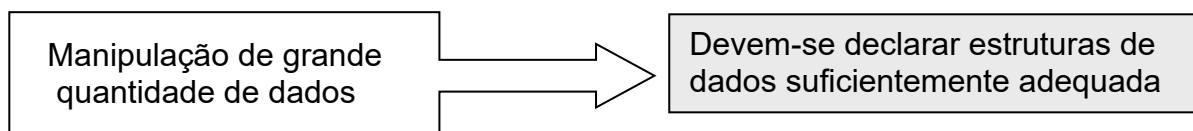
Uma estrutura de dados retrata as relações lógicas existente entre os dados. São arranjos associados à memória principal do computador. Esta memória é organizada em forma de células independentes, chamadas de células de memória.

As estruturas de dados podem ser classificadas em primitivas e não primitivas (também chamadas de complexas ou derivadas).

Primitivas: são aquelas que não podem ser decompostas, por exemplo: inteiro, real, lógica, caractere etc.

Complexas: são aquelas definidas a partir dos dados primitivos, ou seja, são arranjos de dados primitivos, por exemplos: Matrizes, Listas, Pilhas, Filas e Árvores (serão estudadas nesta apostila com exceção de Matrizes).

Estruturas de Dados são utilizadas quando:



Os problemas ocorrem quando precisa criar inúmeras variáveis, o que pode ser impossível estimar o número de dados a ser criado e também pode ser impossível reservar tanta memória. Outro problema também pode ser a forma de manipulação desses dados, causando insegurança e restrição de acesso aos dados.

As características desejáveis na estrutura de dados escolhida é preservar as relações lógicas entre os dados e permitir que algoritmos sejam descritos por rotinas simples e eficientes, possibilitando construção de programas computacionais claros e confiáveis.

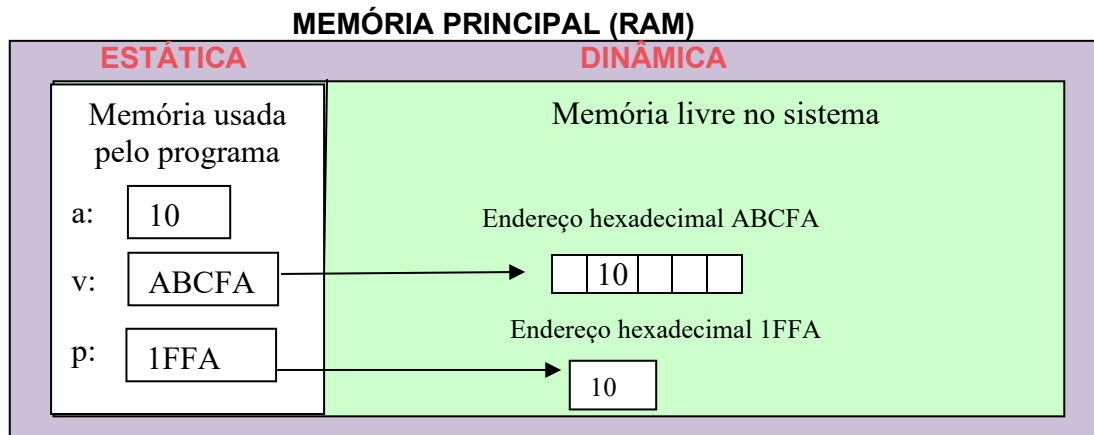
1.2 ALOCAÇÃO ESTÁTICA x DINÂMICA

As variáveis de um programa têm alocação **estática** se a quantidade total de memória utilizada pelos dados é previamente conhecida e definida de modo imutável, no próprio código-fonte do programa. Durante toda a execução, a quantidade de memória utilizada pelo programa não varia. Por outro lado, se o programa é capaz de criar novas variáveis enquanto executa, isto é, se áreas de memória que não foram declaradas no programa passam a existir durante a sua execução, então a alocação de memória é chamada de **dinâmica**.

Exemplo da declaração de variáveis em um programa Java onde usa memória estática e dinâmica:

```
int a = 10;
int [ ] v = new int[5];      (obs: o new instancia/aloca um objeto na memória dinâmica)
v[1] = 10;
int p = new int;
p = 10;
```

¹ **Niklaus E. Wirth** (nascido a 15 de Fevereiro de 1934 em Winterthur, Suíça) é um professor suíço de informática e criador da linguagem de programação Pascal.



Onde:

a = variável primitiva do tipo inteiro.

v = vetor definido com tamanho 5 do tipo inteiro. A variável **v** guarda o endereço da primeira célula reservada no momento da instanciação do vetor.

p = variável do tipo ponteiro, guarda um endereço hexadecimal da memória dinâmica onde está o dado armazenado.

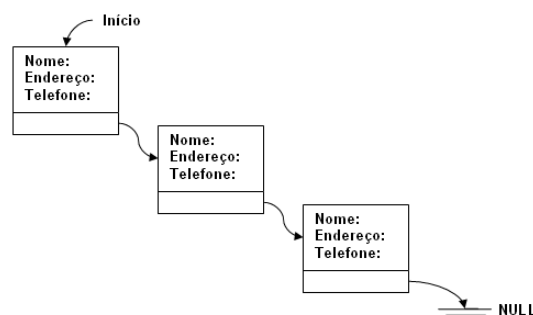
A memória dinâmica pode ser alocada à medida que precisamos e desalocada quando não mais precisamos do dado armazenado, liberando assim memória para ser usada por outros programas de computador. Quem define a alocação e liberação de memória dinâmica é o programador. Já a memória estática é gerenciada pelo sistema operacional.


A Linguagem de Programação Java possibilita uma instanciação rápida e simples, pois ela mesmo faz referência a endereços e alocação dinâmica de memória. Se um objeto não é mais referenciado dentro do programa, o próprio Java trata de liberar os espaços de memória utilizado pelo objeto (dados) usando o Garbage Collector (Coletor de Lixo).

1.3 PONTEIRO ou APONTADOR

Conceito: é uma variável que indica em que ponto (endereço) da memória dinâmica está um dado. Sabe-se que os dados são armazenados na memória do computador, e esta é identificada por endereços numéricos hexadecimais. Sabendo o endereço de um dado ou parte dele, facilita encontrar a sua localização. E os ponteiros são posições de memória que contém o endereço de um dado. Se mais tarde quiser recuperar esse dado, terá acesso à posição que contém esse endereço e poderá recuperá-lo a partir deste.

É como se quisesse ligar para um amigo e não saber o telefone dele de cor, então se pega a agenda de telefones e abre na letra inicial do nome e encontra-se o nome desejado. É assim que funciona o ponteiro. O telefone do amigo é o dado, a agenda de telefones é a memória e a letra inicial é o ponteiro.



NULL (símbolo gráfico: / ou 

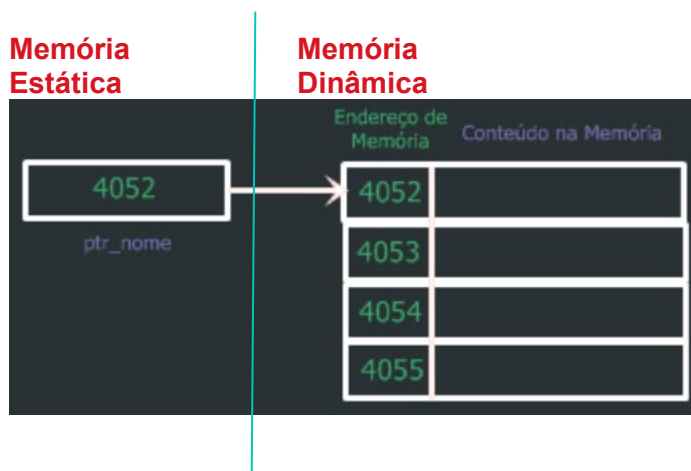
Endereçamento Hexadecimal da Memória RAM

A memória RAM (Random Access Memory) é o local onde o computador armazena os dados e programas que estão sendo usados no momento. Para que o processador possa acessar esses dados, cada célula de memória RAM possui um endereço único. Esse endereço é como o número de uma casa, que permite que o carteiro (o processador) entregue a carta (o dado) no lugar certo.

O sistema de numeração utilizado para endereçar as células de memória RAM é o hexadecimal. Esse sistema, que utiliza 16 símbolos (0-9 e A-F), é mais compacto e fácil de usar do que o sistema decimal (0-9) que usamos no dia a dia.

Exemplo: Imagine uma rua com 256 casas. No sistema decimal, os números das casas iriam de 0 a 255. No sistema hexadecimal, os números das casas iriam de 00 a FF. Ambos os sistemas representam a mesma quantidade de casas, mas o sistema hexadecimal usa menos dígitos.

Logo, o sistema de numeração hexadecimal é fundamental para o funcionamento da memória RAM. Ele permite que o processador acesse os dados de forma rápida e eficiente, garantindo o bom desempenho do computador.



1.4 TIPOS ESTRUTURADOS DE DADOS HETEROGÊNEOS (Registros)

Um tipo estruturado heterogêneo, chamado de registro, é um conjunto de campos, isto é, um conjunto de dados sobre um determinado assunto. Cada campo do registro recebe dados que são armazenados em uma estrutura de dados definida, como um vetor, lista, pilha, árvore, etc.



Para representar uma estrutura do tipo registro em Java, utiliza-se uma **classe**.

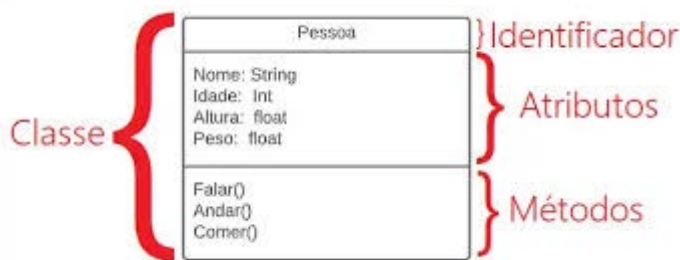
Em programação orientada a objetos (POO), uma **classe** é um modelo ou planta para criar objetos. Ela define os atributos (dados) e métodos (comportamentos) que os objetos daquela classe possuirão. Pense em uma classe como um tipo de dado personalizado que você pode criar.

Principais características de uma classe:

- **Atributos:** São as características ou propriedades de um objeto. Por exemplo, em uma classe "Carro", os atributos podem ser "cor", "marca", "modelo" e "ano".
- **Métodos:** São as ações ou operações que um objeto pode realizar. Usando o exemplo da classe "Carro", os métodos podem ser "ligar()", "acelerar()" e "frear()".
- **Encapsulamento:** Uma classe pode proteger seus dados internos do acesso externo, expondo apenas os métodos necessários para interagir com o objeto. Isso ajuda a manter a integridade dos dados e evita alterações acidentais.
- **Instanciação:** Para criar um objeto a partir de uma classe, você precisa instanciá-la. O objeto resultante é chamado de instância da classe. Por exemplo, você pode criar várias instâncias da classe "Carro", cada uma com seus próprios valores de atributos.

Em resumo, uma classe é um conceito fundamental na POO que permite criar tipos de dados personalizados e organizar o código de forma mais modular e reutilizável. Ela fornece um modelo para criar objetos que compartilham características e comportamentos semelhantes.

Exemplo:



Toda estrutura de dado criada é para guardar dado. Logo, será necessária a criação de uma classe para definir os tipos de dados que serão armazenados nas estruturas de dados.

Nesta disciplina usaremos a classe **Item** onde contém um atributo chamado **chave** que representa o dado que será armazenado na estrutura de dados. Esse atributo é do tipo inteiro, mas poderá ser alterado seu tipo conforme a necessidade de armazenamento. Outros atributos poderão ser incluídos nesta classe Item dependendo da necessidade.

```
public class Item {  
    private int chave;  
    // aqui podem ser declarados outros atributos conforme sua necessidade.  
  
    //Construtor de objetos da classe Item  
    public Item(int ch) {  
        this.chave = ch;  
    }  
    //Modifica o valor do atributo chave  
    public void setChave (int ch){  
        this.chave = ch;  
    }  
    //Faz a leitura do valor do atributo chave  
    public int getChave () {  
        return this.chave;  
    }  
}
```

Capítulo 2 - LISTAS LINEARES

Para representar um grupo de dados, podem-se usar matrizes unidimensionais ou multidimensionais. Ao declarar uma matriz, reserva-se um espaço contíguo na memória para armazenar seus elementos, portanto, deve-se escolher um número máximo de elementos.

No entanto, a matriz não é uma estrutura de dados muito flexível (seu tamanho é constante) e não existe uma maneira simples e barata (computacionalmente) para alterar a dimensão da matriz em tempo de execução. Por outro lado, se o número de elementos que se precisa armazenar na matriz for muito inferior à sua dimensão, estará não utilizando o espaço de memória reservado.

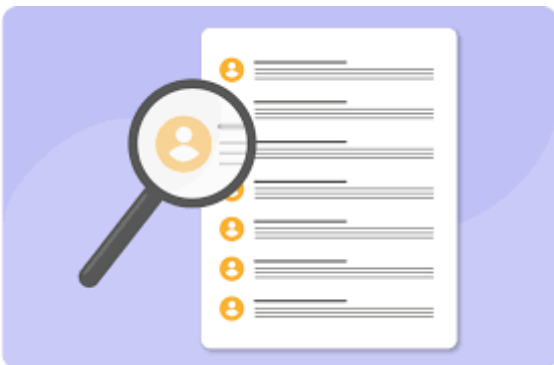
A solução para esse problema é utilizar estruturas de dados que cresçam à medida que precisar de novos elementos e diminuam à medida que precisar retirar elementos armazenados anteriormente. Por isso que usamos as Lista Lineares.

Definição de Lista Linear:

Uma **lista** é uma sequência de um ou mais elementos de um mesmo tipo. É uma estrutura que permite representar um conjunto de dados de forma a preservar a relação de ordem linear existente entre eles e também prever o seu crescimento e decrescimento.

Em outras palavras, a característica fundamental de uma lista linear é o sentido de ordem unidimensional dos elementos que a compõem. Uma ordem que nos permite dizer com precisão onde a coleção inicia-se e onde termina, sem possibilidade de dúvida.

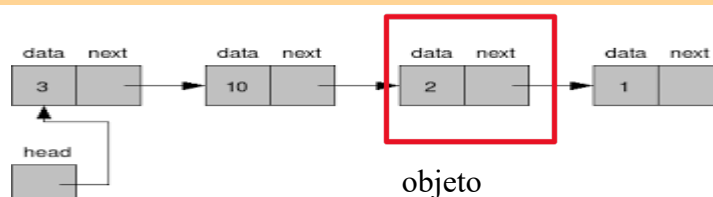
Logo, uma lista é uma coleção de elementos do mesmo tipo dispostos linearmente que podem ou não seguir uma determinada organização.



Exemplos de listas não computacionais:

- Pessoas esperando ônibus no ponto;
- Vagões de trem;
- Letras de uma palavra.
- Lista de Pagamentos:
- Prestação do carro;
- Cartão de crédito;
- Conta de luz;
- Condomínio;
- TV a cabo;
- Prestação do crediário.

Os **objetos** da lista podem ser também chamados de **nós**, **nodos**, **elementos** ou **registros** e possuem um campo contendo o endereço de memória onde encontra o próximo elemento na lista linear.

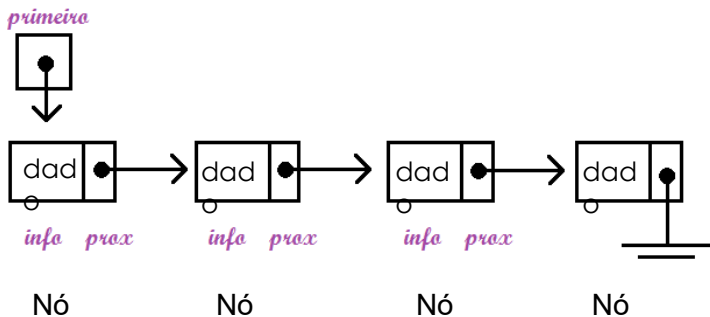


REPRESENTAÇÃO

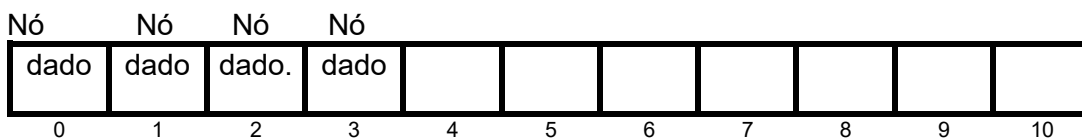
Existem várias formas possíveis de se representar uma lista linear. A escolha depende das operações que serão executadas sobre a lista.

Algumas representações são favoráveis a algumas operações, no sentido de existir um maior esforço computacional para a sua execução. As duas formas de representar as listas lineares são:

- Por **ENCADEAMENTO** dos nós (uso de ponteiros).



- Por **CONTIGUIDADE** dos nós (uso de vetor).



OPERAÇÕES

As operações ou métodos são ações realizadas nos objetos armazenados nas estruturas de dados. Os dados são armazenados para serem pesquisados, inseridos, excluídos e alterados. As operações mais comuns feitas sobre LISTAS LINEARES ENCADEADAS são:

1. **ACESSAR** o determinado nó da lista, para obter e/ou alterar o dado nele contido.
2. **ACESSAR** o k-ésimo nó da lista, para obter e/ou alterar o dado nele contido.
3. **INSERIR** um novo nó após determinado nó da lista;
4. **INSERIR** um novo nó após do k-ésimo nó da lista;
5. **INSERIR** um novo nó antes determinado nó da lista;
6. **INSERIR** um novo nó antes do k-ésimo nó da lista;
7. **INSERIR** um novo nó na lista vazia;
8. **INSERIR** um novo nó após o último nó da lista;
9. **INSERIR** um novo nó em uma lista ordenada;
10. **REMOVER** um determinado nó da lista.
11. **REMOVER** o k-ésimo nó da lista.
12. **REMOVER** o primeiro nó da lista.
13. **REMOVER** o último nó da lista.
14. **REMOVER** o um conjunto específico de nós da lista.
15. **CONCATENAR** duas (ou mais) listas.
16. **PARTIR** uma lista em duas ou mais.
17. Determinar o **TAMANHO** da lista.
18. **CLASSIFICAR** os nós da lista em ordem ascendente (ou descendente) segundo campo chave.
19. **MOSTRAR** o conteúdo dos nós da lista.
20. **INTERCALAR** duas ou mais listas ordenadas segundo o campo chave.

Enfim, são muitas as formas de manipular uma lista linear. É bom lembrar que nem todas as operações citadas podem ser aplicadas nas listas por contiguidade.

APLICAÇÕES

Listas são adequadas para aplicações onde não é possível prever a demanda por memória, permitindo a manipulação de quantidades imprevisíveis de dados, de formato também imprevisível. Listas são úteis em aplicações tais como:

- Manipulação de dados
- Gerenciamento de memória
- Simulação e Compiladores
- Sistemas Operacionais
- Banco de Dados
- Linguagem de programação

Como já foi citado, as listas lineares são estruturas de dados sequenciais, nas quais os elementos são armazenados em uma ordem específica. Cada elemento possui um único predecessor (exceto o primeiro) e um único sucessor (exceto o último).

Vamos ver alguns exemplos de listas lineares e suas aplicações:

1. **Arrays:**

- **Aplicações:** Armazenamento de coleções de dados do mesmo tipo, como números em cálculos matemáticos, registros de funcionários em um banco de dados, pixels de uma imagem.
- **Características:** Acesso rápido a elementos por meio de índices, tamanho fixo (em linguagens como C e Java).

2. **Listas encadeadas:**

- **Aplicações:** Implementação de pilhas e filas, listas de tarefas, representação de listas de reprodução de música, gerenciamento de memória.
- **Características:** Alocação dinâmica de memória, inserção e remoção eficientes de elementos, acesso sequencial aos elementos.

3. **Pilhas (Stacks):**

- **Aplicações:** Funções de chamada em programas, histórico de navegação em browsers, desfazer/refazer em editores de texto.
- **Características:** Estrutura LIFO (Last-In, First-Out), onde o último elemento inserido é o primeiro a ser removido.

4. **Filas (Queues):**

- **Aplicações:** Filas de espera em sistemas, impressão de documentos, processamento de tarefas em sistemas operacionais.
- **Características:** Estrutura FIFO (First-In, First-Out), onde o primeiro elemento inserido é o primeiro a ser removido.

Outras aplicações de listas lineares:

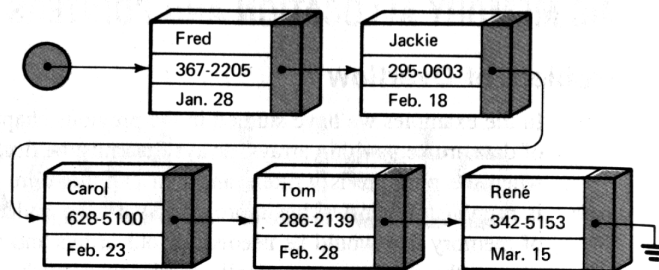
- **Tabelas Hash:** Utilização de listas encadeadas para tratar colisões.
- **Grafos:** Representação de listas de adjacência para modelar relacionamentos entre vértices.
- **Algoritmos de ordenação:** Muitos algoritmos utilizam listas lineares para armazenar os dados a serem ordenados.
- **Compiladores:** Análise léxica e sintática de código fonte utilizam listas lineares para representar tokens e estruturas gramaticais.

As listas lineares são estruturas de dados versáteis e fundamentais na ciência da computação, com aplicações em diversas áreas. A escolha da estrutura linear mais adequada depende das necessidades específicas de cada problema.

Capítulo 3 - LISTA LINEAR SIMPLEMENTE ENCADEADA

Vetores, em geral, utilizam memória pré-alocada, estática e pouco flexível para inclusões e remoções de novos elementos. Vetores dinâmicos também dificultam a manutenção de elementos. Uma alternativa é alocação de pequenas porções de memória ligadas entre si.

Listas simplesmente encadeadas ou **Listas Simplesmente Ligadas** (linked lists) são coleções de objetos ordenados de forma que cada objeto guarda, além de dados, informação sobre o endereço do próximo objeto da lista, formando uma cadeia de objetos.



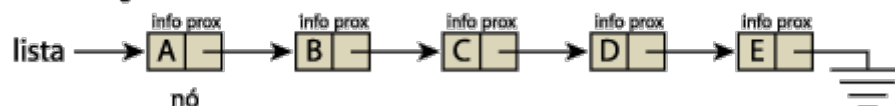
Propriedades

- A lista pode ter 0 ou infinitos itens;
- Permite o crescimento dinâmico do comprimento da lista linear;
- Diminui o esforço computacional das operações de inserção e remoção de elementos na lista.

A implementação desta lista é através de apontadores, cada item da lista é encadeado com o seguinte através de uma variável do tipo Apontador. Este tipo de implementação permite utilizar posições não contíguas de memória.

Dessa forma, a lista pode ser distribuída com um grupo de vários pequenos blocos, interligados por ponteiros, que juntos formam uma só lista. Se cada bloco contiver cinco posições, quatro vão ser usadas para armazenar os elementos e uma posição (a última) será como um ponteiro que guardará o endereço do próximo elemento da lista (que está em outro bloco).

Uma lista ligada linear



Para localizar o primeiro elemento da lista, reserva-se uma posição da memória, na qual é guardado o endereço do primeiro elemento. Esta posição é chamada de ponteiro para o início da lista.

Para consultar a lista, inicia-se pela posição indicada pelo ponteiro para o início da lista, percorrendo os elementos do primeiro bloco e continuando pela posição indicada pelo ponteiro da última posição do primeiro bloco, seguindo para o segundo bloco e assim em diante até que chegue o final da lista, indicado por um ponteiro vazio (um padrão especial de bits que indicam que não existe nenhum elemento adicional na lista).

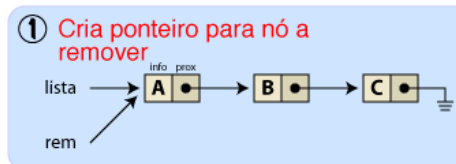
Vantagens

- Não requer movimentação de nós nas operações de inserção e eliminação (como na lista por contiguidade);
- Apenas os ponteiros são alterados (lembre-se que cada registro pode conter campos complexos);
- Não é necessário saber, anteriormente, o número máximo de elementos que uma lista pode ter, o que implica em não desperdiçar espaço na Memória Principal do computador;
- Tamanho máximo para a lista é o tamanho da memória livre do computador.

Desvantagens

- O acesso não é indexado, para acessar um k-ésimo nó tem que percorrer do primeiro elemento até chegar ao desejado;
- Aumento do tempo de execução, o qual é gasto na obtenção de espaço de memória;
- Reserva de espaço para armazenar os apontadores (campo PROX);
- Necessário gerenciar os ponteiros PROX;
- A manipulação torna-se mais "perigosa" uma vez que, se o encadeamento (ligação) entre elementos da lista for mal feito, toda a lista pode ser perdida.

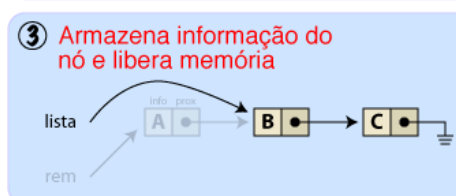
3.1 ELIMINAÇÃO E INSERÇÃO DE ELEMENTOS NA LISTA



Para eliminar um elemento, este pode ser removido alterando-se um único ponteiro. Isto é feito modificando-se o ponteiro, de forma que passe a apontar o elemento que segue na lista. Assim, quando a lista for percorrida, o elemento removido será ignorado.



Se o elemento a ser retirado é o primeiro da lista, deve-se fazer com que o ponteiro para o início da lista passe a apontar para o segundo elemento, então se pode liberar o espaço que pertencia ao elemento retirado.

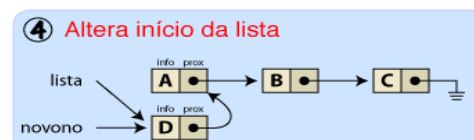
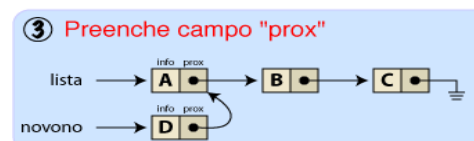
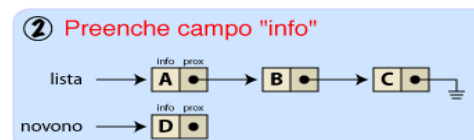
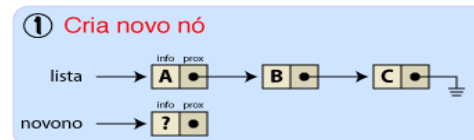


Se o elemento a ser retirado estiver no meio da lista, deve-se fazer com que o ponteiro anterior passe a apontar para o elemento seguinte, liberando assim o espaço ocupado pelo elemento retirado.

Para implementar elementos numa lista, deve-se primeiro separar um espaço na memória e então preenchê-lo com os elementos.

Se o elemento a ser inserido for o primeiro da lista, deve-se fazer com que o ponteiro que indicava o primeiro elemento passe a apontar para o elemento inserido.

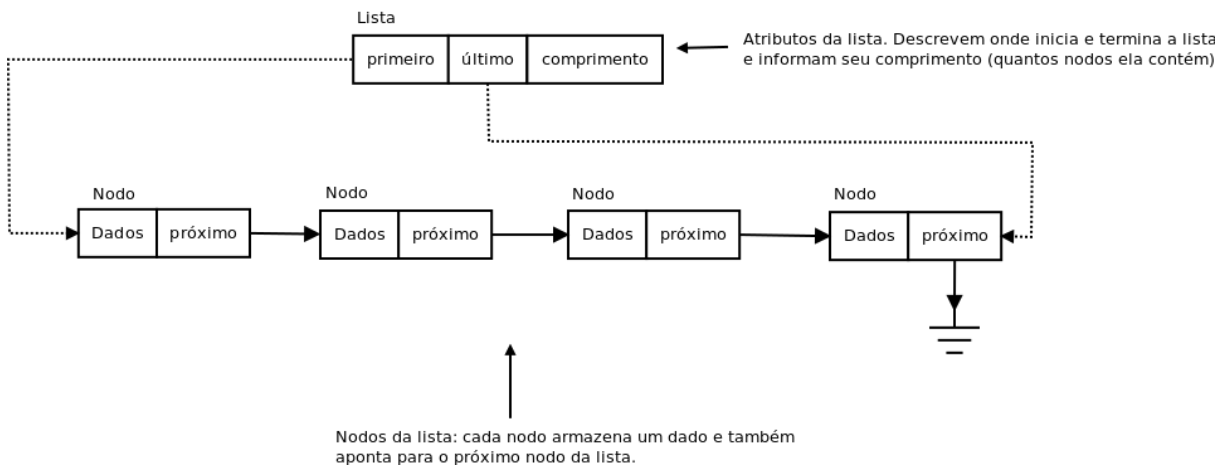
Se o elemento a ser inserido for pertencer ao meio da lista, por exemplo, o terceiro elemento, deve-se fazer com que, nesse caso, o ponteiro do segundo elemento passe a apontar para o novo elemento e o ponteiro do novo elemento passe a apontar para o que antes era o terceiro elemento e agora é o quarto elemento. Esse procedimento também serve para inserir elementos no fim da lista.



3.2 LISTA SIMPLEMENTE ENCADEADA COM DESCRITOR

Para otimizar o gerenciamento da lista, introduzimos o conceito de descritor. O descritor é um nó especial que serve como ponto de referência para a lista. Ele contém informações cruciais, como:

- **Ponteiro para o primeiro nó:** Facilita o acesso ao início da lista.
- **Contagem de elementos:** Mantém o controle do tamanho da lista, evitando iterações desnecessárias.
- **Ponteiro para o último nó:** Permite a inserção eficiente de elementos no final da lista.

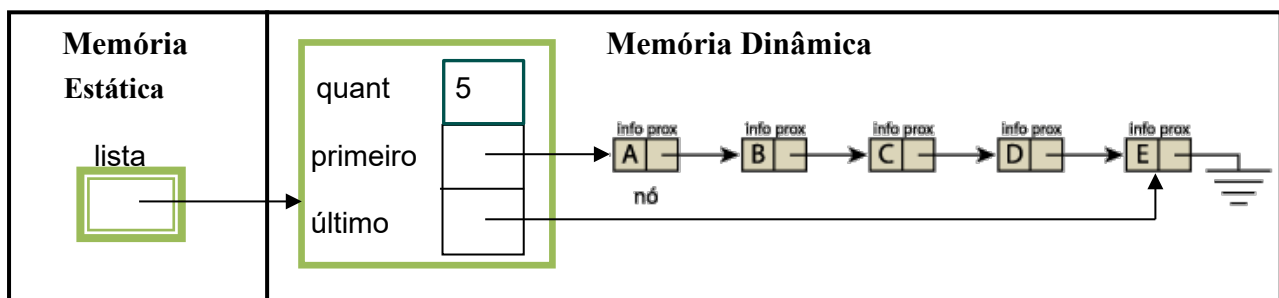


Vantagens do Descritor

- **Acesso rápido ao início e fim:** O descritor agiliza operações que envolvem o início ou o fim da lista, como inserção e remoção.
- **Controle do tamanho:** A contagem de elementos no descritor elimina a necessidade de percorrer toda a lista para determinar seu tamanho.
- **Código mais organizado:** O descritor centraliza informações importantes, tornando o código mais legível e fácil de manter.
- **Eficiência em inserções e remoções:** O descritor torna mais rápidas as inserções e remoções no começo e final da lista, pois o código não precisa percorrer todo o conteúdo para encontrar o primeiro ou último elemento.

Funcionamento Detalhado

Cada nó da lista, exceto o descritor, contém um dado e um ponteiro para o próximo nó. O último nó da lista aponta para NULL, indicando o final da sequência. O descritor, por sua vez, armazena os ponteiros para o primeiro e o último nó, além da contagem de elementos.



Aplicações Práticas

Listas lineares simplesmente encadeadas com descritor são utilizadas em diversas aplicações, como:

- **Implementação de pilhas e filas:** A estrutura simplificada facilita a criação dessas estruturas de dados abstratas.
- **Gerenciamento de listas de tarefas:** A inserção e remoção eficientes de elementos tornam a estrutura ideal para organizar tarefas.
- **Representação de listas de reprodução:** A flexibilidade da estrutura permite a criação de listas de reprodução dinâmicas.
- **Gerenciamento de memória:** A alocação dinâmica de memória facilita o gerenciamento eficiente de recursos.

Em resumo, a lista linear simplesmente encadeada com descritor é uma estrutura de dados poderosa e flexível, que oferece organização eficiente de dados e otimização de operações.

3.3 OPERAÇÕES BÁSICAS DA LISTA SIMPLESMENTE ENCADEADA

3.3.1. Classe da declaração do nó:

```
package simplesmente;
import dados.Item;
public class No {
    private Item info; // a declaração do tipo Item está no capítulo 1
    private No prox;

    public No (Item elem){ // a variável elem contém os dados armazenados
        this.info = elem;
        this.prox = null; // esta linha é opcional, pois o prox é automaticamente
                          //definido como null
    }
    public Item getInfo (){
        return this.info;
    }
    public void setInfo(Item elem){
        this.info = elem;
    }
    public No getProx(){
        return this.prox;
    }
    public void setProx(No novoNo){
        this.prox = novoNo;
    }
}
```

3.3.2 Classe da declaração da Lista Simplesmente Encadeada:

```
package simplesmente;
import dados.Item;
public class ListaSimples {
    private No prim;
    private No ult;
    private int quantNos;
    //construtor da classe
    public ListaSimples(){
        this.prim = null;
        this.ult = null;
        this.quantNos = 0;
    }
    public int getQuantNos(){
        return this.quantNos;
    }
    public No getPrim(){
        return this.prim;
    }
    public No getUlt(){
        return this.ult;
    }
}
```

```
}
public void setQuantNos(int novoValor){
    this.quantNos = novoValor;
}
public void setPrim(No novoNo){
    this.prim = novoNo;
}
public void setUlt(No novoNo){
    this.ult = novoNo;
}
public boolean eVazia (){
    return (this.prim == null);
}
//insere um novo nó no final da lista ou se a lista estiver vazia, insere o primeiro nó na lista
public void inserirUltimo (Item elem){
    No novoNo = new No (elem);
    if (this.eVazia()){
        this.prim = novoNo;
    } else {
        this.ult.setProx(novoNo);
    }
    this.ult = novoNo;
    this.quantNos++;
}
//retorna o endereço do nó que está contendo o valor a ser procurado.
public No pesquisarNo (int chave){
    No atual = this.prim;
    while ((atual != null) && (atual.getInfo().getChave() != chave)){
        atual = atual.getProx();
    }
    return atual;
}
public boolean removerNo(int x){
    if (this.eVazia()){
        return false;
    }else{
        if (this.prim.getInfo().getChave()==x){
            if (this.prim.getProx()==null){ //se for único nó da lista
                this.ult = null;
            }
            //remover o primeiro nó da lista
            this.prim = this.prim.getProx();
        }else{
            No atual = this.prim;
            while ((atual.getProx()!=null)&&
                (atual.getProx().getInfo().getChave()!=x)){
                atual = atual.getProx();
            }
            if (atual.getProx()==null){//não achou o valor na lista
```

```
        return false;
    }else{
        if (atual.getProx()==this.ult){ //se for o último nó
            atual.setProx(null);
            this.ult = atual;
        }else{ //remove o nó no meio da lista
            atual.setProx(atual.getProx().getProx());
        }
    }
}
this.quantNos--;
return true;
}
}
//mostra todo o conteúdo da lista
public String toString(){
    String msg = "";
    No atual = this.prim;
    while (atual != null){
        msg += atual.getInfo().getChave()+"\n";
        atual = atual.getProx();
    }
    return msg;
}
}
```

3.3.3. Aplicativo para manipular a lista simplesmente encadeada:

```
package simplesmente;
import java.util.Scanner;
import dados.Item;

public class MenuPrincipal {
    static Scanner scan = new Scanner (System.in);
    public static void main(String[] args) {
        //instancia a lista vazia
        ListaSimples lista = new ListaSimples();
        char opcao;
        int valor;
        Item item;
        do{
            System.out.println("escolha uma opção:\n"
                + "1. inserir nó no final da lista\n"
                + "2. pesquisar se um determinado valor\n"
                + "3. remover um determinado nó da lista\n"
                + "4. mostrar conteúdo da lista\n"
                + "5. sair do programa");
            opcao = scan.next().charAt(0);
            switch (opcao){
```



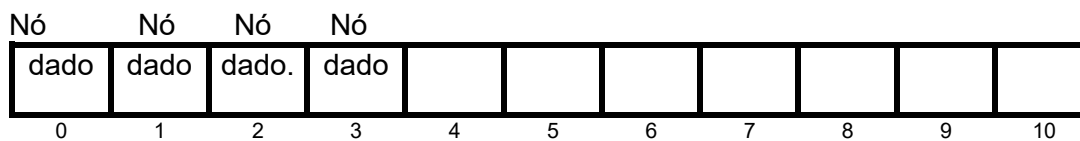
```
        case '1':
            System.out.println("digite um valor inteiro:");
            valor = scan.nextInt();
            item = new Item(valor);
            lista.inserirUltimo(item);
            System.out.println("operação realizada com sucesso!");
            break;
        case '2':
            System.out.println("digite o valor inteiro a ser procurado na lista:");
            valor = scan.nextInt();
            if (lista.pesquisarNo(valor)!=null){
                System.out.println("o valor está na lista");
            }else{
                System.out.println("o valor não está na lista");
            }
            break;
        case '3':
            System.out.println("digite o valor inteiro a ser removido da lista:");
            valor = scan.nextInt();
            if (lista.removerNo(valor)){
                System.out.println("removido com sucesso!");
            }else{
                System.out.println("o valor não está na lista");
            }
            break;
        case '4':
            System.out.println("Mostrar a lista:\n"+lista.toString());
            break;
        case '5':
            System.out.println("fim");
            break;
        default:
            System.out.println("opção inválida");
    }
}while (opcao!='4');
}
```

Capítulo 4 - LISTA LINEAR POR CONTIGUIDADE

É uma técnica para armazenar uma lista completa em um único bloco de posições da memória. Essa organização é típica de sistemas de armazenamento de lista em vetor.

Sabe-se que uma lista contígua tem tamanho constante, e suponha-se que queira eliminar um elemento dessa lista, o primeiro elemento por exemplo, e a desvantagem desse tipo de organização é que, com a eliminação desse primeiro elemento, devem-se deslocar todos os elementos restantes da lista para o começo da lista, suprimindo o espaço vazio deixado pela eliminação do primeiro elemento. Essa é uma das desvantagens desse tipo de estrutura. Uma outra desvantagem é que se quiser adicionar elementos a essa lista, pode ser que haja falta de espaço na memória.

Exemplo de uma representação de uma LISTA LINEAR POR CONTIGUIDADE contendo 4 elementos:



primeiro = 0

último = 4-1

Comentários

A implementação de listas lineares através de vetores tem como vantagem a economia de memória, após os apontadores são implícitos numa estrutura. Como desvantagens citam:

- (I) O custo para inserir ou retirar nós que pode causar um deslocamento de todos os nós no pior caso.
- (II) Em aplicações em que não existe previsão de vetores sobre crescimento da lista, a utilização de vetores pode ser problemática.

4.1 COMPARAÇÃO ENTRE A REPRESENTAÇÃO POR CONTIGUIDADE E POR ENCADEAMENTO

1 - A representação por encadeamento (uso de apontadores) permite a superposição de tabelas partilhando registros comuns com economia de espaço, ou seja, não há necessidade de armazenar um mesmo registro em diferentes posições de memória.

2 – As operações em listas encadeadas são mais simples eliminação e inserção de um nó. Essas operações seriam bem mais demoradas nas listas representadas contiguamente, uma vez que provocaram movimentação de parte da lista.

3 – O tempo de acesso do k-ésimo nó de uma lista representada por contiguidade é uma constante, mas é proporcional a k numa lista encadeada.

4 – A representação por encadeamento torna mais simples a reunião de várias listas numa única, ou a partição de uma lista em várias outras.

5 – O esquema encadeado permite a representação de estruturas mais complexas: pode-se ter um número variável de listas de comprimento variável, e cada nó de uma lista pode ser o ponto de partida para outra, os mesmos nós podem ser ligados em várias ordens correspondendo a diferentes listas, etc.

6 – Graças ao acesso direto, o caminhamento sucessivo através dos nós de uma lista representada por contiguidade é geralmente mais rápido do que uma lista encadeada.

7 – A maior desvantagem da representação por encadeamento é a utilização de memória extra para armazenar os apontadores.

Será sempre necessária uma análise cuidadosa do problema para que se conclua qual o tipo de representação é mais adequado para o problema específico.

4.2 OPERAÇÕES BÁSICAS DA LISTA POR CONTIGUIDADE

Classe da declaração da lista:

```
public class ListaContig {
    private int fim;
    private Item [] info; // o tipo Item está declarado no capítulo 1

    public ListaContig(int qte){
        this.fim = 0;
        this.info = new Item [qte];
    }
    public Item getInfo(int i){
        return this.info[i];
    }
    public void setInfo(int i, Item elem){
        this.info[i]=elem;
    }
    public int getFim(){
        return this.fim;
    }
    public void setFim(int _fim){
        this.fim = _fim;
    }
    public boolean eVazia (){
        return (this.fim == 0);
    }
    public boolean eCheia (){
        return (this.fim == this.info.length);
    }
    //retorna verdadeiro se conseguiu inserir o novo nó no final na lista.
    public boolean inserirUltimo (Item elem){
        if (this.eCheia()){
            return false;
        } else {
            this.info[this.fim]= elem;
            this.fim++;
            return true;
        }
    }
    //retorna a posição do nó caso ele seja encontrado, caso contrário, retorna
    //o valor do this.fim simbolizando que não foi encontrado.
    public int pesquisarNo (int chave){
        int i = 0;
        while ((i < this.fim) && (this.info[i].getChave() != chave)){
            i++;
        }
        return i;
    }
    //retorna verdadeiro se conseguiu remover um nó específico.
    public boolean removerNo (int chave){
        int i = 0;
        while ((i < this.fim) && (this.info[i].getChave() != chave)){
            i++;
        }
        if (i == this.fim){
```

```
        return false;
    }else{
        for (int j = i; j < this.fim-1 ;j++){
            this.info[j] = this.info[j+1];
        }
        this.fim--;
        return true;
    }
}

//retorna uma String com todo o conteúdo da lista.
public String toString(){
    String msg="";
    for (int i=0; i < this.fim; i++){
        msg += this.info[i].getChave()+"\n";
    }
    return msg;
}
}
```

Aplicativo para manipular a lista contigua:

```
package listaContig;
import java.util.Scanner;
import dados.Item;
public class MenuPrincipal {
    static Scanner scan = new Scanner(System.in);
    public static void main(String[] args) {
        System.out.println("digite o tamanho máximo da lista");
        int tam = scan.nextInt();
        ListaContig lista = new ListaContig(tam);
        int valor;
        char opcao;
        do {
            System.out.println("Escolha uma Opção:\n"+
                "1. Inserir Nó no final\n"+
                "2. Localizar Nó\n"+
                "3. Excluir Nó\n"+
                "4. Exibir lista\n"+
                "5. Sair");
            opcao = scan.next().charAt(0);
            switch (opcao){
                case '1':
                    System.out.println("Inserir um Valor no final da lista\n"+
                        "Digite um valor");
                    valor = scan.nextInt();
                    if (! lista.inserirUltimo(new Item(valor))){
                        System.out.println("Lista está cheia");
                    }
                    break;
                case '2':
                    if (lista.eVazia()){
                        System.out.println("A lista está vazia");
                    }
                    else{
                        System.out.println("Digite o valor para localizar");
                        valor = scan.nextInt();
                        int pesqNo = lista.pesquisarNo(valor);
                        if (pesqNo == lista.getFim()){
                            System.out.println("Não encontrado\n");
                        }else{
                            System.out.println("Está na posição "+pesqNo);
                        }
                    }
                }
            }
        }
    }
}
```

```
        break;
    case '3':
        if (lista.eVazia()){
            System.out.println("A lista está vazia");
        }
        else {
            System.out.println("Digite um valor para excluí-lo:");
            valor = scan.nextInt();
            if (lista.removerNo(valor)){
                System.out.println("remoção efetuada");
            }
            else{
                System.out.println("remoção não efetuada,"+
                    " valor não encontrado");
            }
        }
        break;
    case '4':
        if (lista.eVazia()){
            System.out.println("A lista está vazia");
        }
        else{
            System.out.println("Exibir a lista"+lista.toString());
        }
        break;
    case '5':
        System.out.println("fim do programa");
        break;
    default:
        System.out.println("opção invalida, tente novamente");
    }
} while (opcao!='5');
System.exit(0);
}
```

Capítulo 5 - LISTA LINEAR DUPLAMENTE ENCADEADA

Se em uma lista encadeada unicamente, tivermos um apontador apontando para um determinado nó, podemos continuar a percorrer a lista sequencialmente (da esquerda para a direita). Porém se mudarmos de ideia e resolvermos voltar a apontar para o nó anterior, ficaríamos impossibilitados pelo fato de que não existe um ponteiro apontando para o nó anterior. Neste caso, teremos que, inicializar novamente o percurso a partir do início da lista, até apontarmos para o nó desejado. Para contornar este problema, iremos construir um registro (nó) que irá conter 2 apontadores, que irão apontar para o nó anterior e para o nó posterior respectivamente.

Então, cada nó possui duas referências:

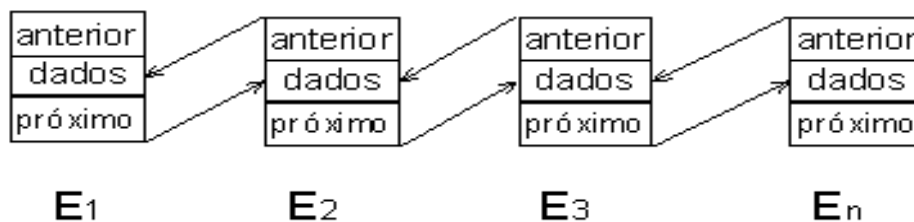
- a primeira aponta para o nó predecessor;
- a segunda aponta para o nó sucessor.



Com esta nova estrutura podemos percorrer a lista e voltarmos ao nó anterior, caso seja necessário. Entretanto, as operações básicas (inserção e remoção) tornam-se mais detalhadas.

Exemplo de uma Lista Duplamente Encadeada com n nós:

Nó



A lista linear duplamente encadeada com descritor é uma estrutura de dados sofisticada que oferece flexibilidade e eficiência no gerenciamento de dados sequenciais. Diferente da lista simplesmente encadeada, cada nó nesta estrutura possui dois ponteiros: um para o próximo nó e outro para o nó anterior, permitindo a navegação bidirecional.

Descritor: O Ponto de Controle

O descritor, um nó especial, desempenha um papel crucial na organização e controle da lista. Ele armazena informações importantes, como:

- **Ponteiro para o primeiro nó:** Facilita o acesso ao início da lista.
- **Ponteiro para o último nó:** Agiliza operações no final da lista.
- **Contagem de elementos:** Mantém o controle do tamanho da lista, evitando iterações desnecessárias.

Navegação Bidirecional: A Vantagem Principal

A principal vantagem da lista duplamente encadeada é a capacidade de navegar em ambas as direções. Isso torna mais fácil realizar operações como:

- **Inserção e remoção de elementos:** Acesso direto aos nós anterior e posterior agiliza essas operações.

- **Busca de elementos:** A navegação bidirecional torna a busca mais eficiente, especialmente em listas grandes.
- **Implementação de estruturas complexas:** A flexibilidade da estrutura facilita a criação de outras estruturas de dados, como pilhas e filas duplas.

Funcionamento Detalhado

Cada nó da lista, exceto o descritor, contém um dado, um ponteiro para o próximo nó e um ponteiro para o nó anterior. O primeiro nó aponta para NULL no ponteiro anterior, e o último nó aponta para NULL no ponteiro seguinte. O descritor armazena os ponteiros para o primeiro e último nó, além da contagem de elementos.

Aplicações Práticas

Listas lineares duplamente encadeadas com descritor são utilizadas em diversas aplicações, como:

- **Editores de texto:** A navegação bidirecional facilita a edição de documentos.
- **Navegadores web:** O histórico de navegação pode ser implementado com essa estrutura.
- **Reprodutores de música:** Listas de reprodução com navegação para frente e para trás.
- **Sistemas de cache:** A remoção de elementos em qualquer posição é facilitada.

Em resumo, a lista linear duplamente encadeada com descritor é uma estrutura de dados poderosa que oferece navegação bidirecional e controle eficiente, tornando-a ideal para aplicações que exigem flexibilidade e desempenho.

5.1 OPERAÇÕES BÁSICAS DA LISTA DUPLAMENTE ENCADEADA

Classe da declaração do tipo nó da lista duplamente encadeada:

```
public class NoDupla {
    private Item info;           // o tipo Item está declarado no capítulo 1
    private NoDupla prox;
    private NoDupla ant;

    public NoDupla (Item elem){
        this.info = elem;
        this.prox = null;
        this.ant = null;
    }
    public Item getInfo () {
        return this.info;
    }
    public NoDupla getProx() {
        return this.prox;
    }
    public NoDupla getAnt() {
        return this.ant;
    }
    public void setProx(NoDupla novoNo) {
        this.prox = novoNo;
    }
    public void setAnt(NoDupla novoNo) {
        this.ant = novoNo;
    }
}
```

Classe da declaração da lista duplamente encadeada:

```
public class ListaDupla {
    private NoDupla prim;
    private NoDupla ult;
    private int quantNos;
```

```
public ListaDupla() {
    this.prim = null;
    this.ult = null;
    this.quantNos = 0;
}
public int getQuantNos() {
    return this.quantNos;
}
public NoDupla getPrim() {
    return this.prim;
}
public NoDupla getUlt() {
    return this.ult;
}
public void setQuantNos(int valorNovo) {
    this.quantNos = valorNovo;
}
public void setPrim(NoDupla novoNo) {
    this.prim = novoNo;
}
public void setUlt(NoDupla novoNo) {
    this.ult = novoNo;
}
public boolean eVazia () {
    return (this.prim == null);
}
//insere um novo nó no final da lista ou se a lista estiver vazia, insere
// o primeiro nó na lista
public void inserirUltimo (Item elem) {
    NoDupla novoNo = new NoDupla (elem);
    if (this.eVazia())
        this.prim = novoNo;
    else {
        novoNo.setAnt(this.ult);
        this.ult.setProx(novoNo);
    }
    this.ult = novoNo;
    this.quantNos++;
}
//retorna o endereço do nó que está contendo o valor a ser procurado.
public NoDupla pesquisarNo (int chave) {
    NoDupla atual = this.prim;
    while ((atual != null) && (atual.getInfo().getChave() != chave))
        atual = atual.getProx();
    return atual;
}
//remove um determinado nó em qualquer posição na lista.
public boolean removerNo (int chave) {
    NoDupla atual = this.prim;
    while ((atual != null) && (atual.getInfo().getChave() != chave)) {
        atual = atual.getProx();
    }
    if (atual == null)
        return false;
    else
        if (atual == this.prim) {
            this.prim = prim.getProx();
            if (this.prim == null) //se a lista tem somente um nó
                this.ult=null;
            else
                this.prim.setAnt(null);
        }
        else
```



```
        if (atual == this.ult) {
            this.ult = this.ult.getAnt();
            this.ult.setProx(null);
        }
        else {
            atual.getProx().setAnt(atual.getAnt());
            atual.getAnt().setProx(atual.getProx());
        }
        this.quantNos--;
        return true;
    }
    public String toString(){
        String msg="";
        NoDupla atual = this.prim;
        while (atual != null){
            msg += atual.getInfo().getChave()+"\n";
            atual = atual.getProx();
        }
        return msg;
    }
}
```

Aplicativo para manipular a lista duplamente encadeada:

```
public class MenuPrincipalDupla {
    static Scanner scan = new Scanner(System.in);
    public static void main(String[] args) {
        //instancia uma nova lista duplamente encadeada.
        ListaDupla lista = new ListaDupla();
        int valor;
        NoDupla pesqNo = null;
        char opcao;
        do {
            opcao = menu();
            switch (opcao){
                case '1':
                    System.out.println("Digite um valor");
                    valor = scan.nextInt();
                    lista.inserirUltimo(new Item(valor));
                    break;
                case '2':
                    System.out.println("Digite um valor");
                    valor = scan.nextInt();
                    pesqNo = lista.pesquisarNo(valor);
                    if (pesqNo==null)
                        System.out.println("o "+valor+" não foi achado");
                    else
                        System.out.println("o "+pesqNo.getInfo().getChave()+
                            " foi achado na lista");
                    break;
                case '3':
                    System.out.println("Digite um valor");
                    valor = scan.nextInt();
                    if (lista.removerNo(valor))
                        System.out.println("remoção efetuada");
                    else
                        System.out.println("remoção não efetuada");
                    break;
                case '4':
```

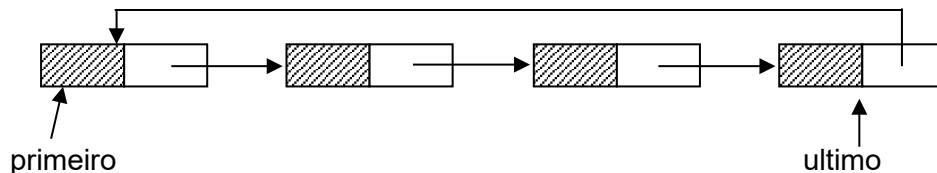
```
        System.out.println("Exibir a lista\n"+lista.toString());
        break;
    case '5':
        System.out.println("fim do programa");
    }
} while (opcao != '5');
System.exit(0);
}
public static char menu(){
    System.out.println("Escolha uma Opção:\n"+
        "1. Inserir Nó no fim\n"+
        "2. Localizar Nó\n"+
        "3. Excluir Nó\n"+
        "4. Exibir lista\n"+
        "5. Sair");
    return scan.next().charAt(0);
}
}
```

Capítulo 6 - LISTAS LINEARES CIRCULARES

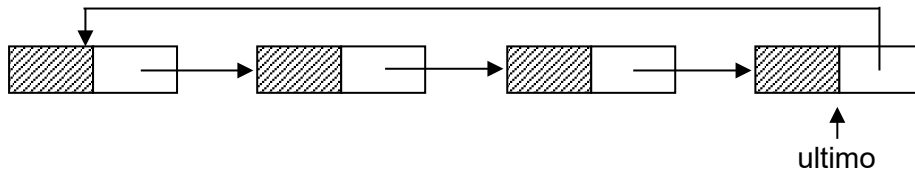
As estruturas a seguir, são apresentadas para fins de conhecimento da sua existência.

6.1 SIMPLEMENTE ENCADEADA

Numa lista com encadeamento circular, ao invés do campo do último nó armazenar um endereço nulo, ele armazena o endereço do primeiro nó.



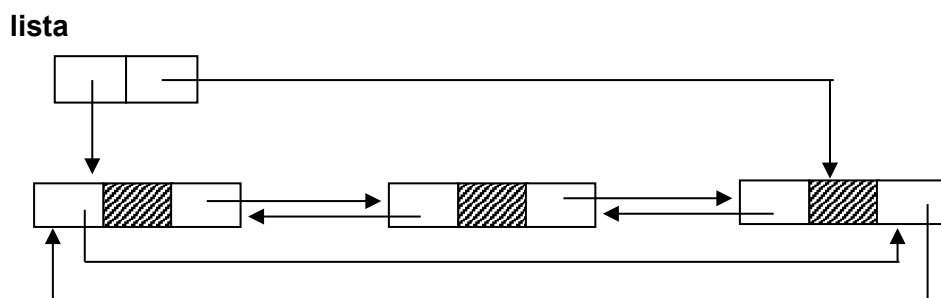
Como o endereço do primeiro nó pode ser facilmente obtido através do campo do último nó, a variável ponteiro para uma lista circularmente encadeada guarda não o endereço do primeiro, mas sim o do último nó da lista.



A grande vantagem em se usar encadeamento circular é que pode-se acessar rapidamente tanto o primeiro quanto o último nó da lista. Na implementação não circular, acessar o último elemento requer a passagem por todos os elementos da cadeia, um a um, até atingir o último, quando não se tem o ponteiro para o último.

6.2 DUPLAMENTE ENCADEADA

O último nó aponta para o primeiro nó, e o primeiro nó aponta para o último nó.



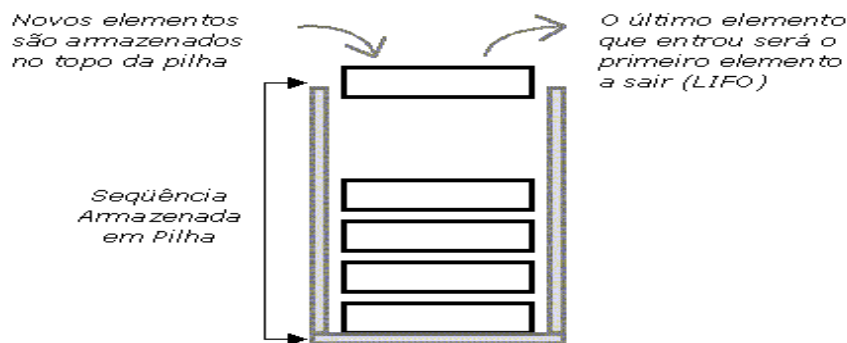
Note que em uma Lista Duplamente Encadeada Circular, cada nó satisfaz:

$$(p^{dir})^{esq} = p = (p^{esq})^{dir}$$

Com a organização proposta acima, a implementação da operação de remoção do nó da direita não necessitada mais do caminhamento linear sobre a lista.

Capítulo 7 - PILHA

Pilha é um tipo especial de lista linear em que todas as operações de inserção e remoção são realizadas numa mesma extremidade, denominada TOPO. Cada vez que um novo elemento deve ser inserido na pilha, ele é colocado no seu topo; e em qualquer momento, apenas aquele posicionado no topo da pilha pode ser removido. Devido a essa forma de trabalhar, os elementos são sempre removidos numa ordem inversa a que foram inseridos, de modo que o último elemento que entra é exatamente o primeiro que sai.



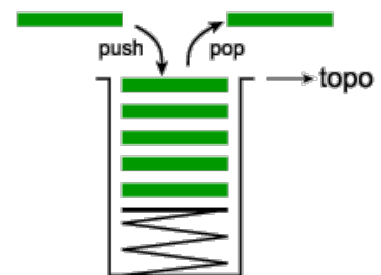
Ao inserir um item numa pilha, este é armazenado no topo da pilha, assim chamada a extremidade por onde são executadas as operações de empilhamento (inserção) e desempilhamento (remoção). Da mesma forma o primeiro item a ser desempilhado é o que está no topo da pilha. Este fato faz com que para se ter acesso a um elemento no meio da pilha é necessário que se desempilhem todos os elementos até que o desejado esteja na posição referente ao topo da pilha.

Para mostrar que o acesso a uma pilha é restrito apenas ao elemento mais ao topo, vamos comparar esta situação com uma situação envolvendo uma estrutura de pilhas com a qual estamos bem familiarizados. Por exemplo, uma pilha de livros em uma mesa. Esta organização gera por si mesma, operações tais como inserção e remoção de livros em cima da pilha. Ao colocarmos um livro em uma pilha de livros, automaticamente colocamos o livro por cima da pilha, da mesma forma retiramos o livro que está no topo da pilha. Se tentarmos retirar um livro, cuja localização está no meio da pilha, terá problemas, podendo até mesmo destruir, ou desmontar toda a pilha de livros.

Propriedade

LIFO ("Last-in, First-out") = ÚLTIMO a entrar, PRIMEIRO a sair.

O processo de inserção é chamado de empilhamento ou *push* (do inglês, que significa empilhar) e o processo de remoção é chamado de desempilhamento ou *pop* (do inglês, que significa desempilhar).



Representação

A pilha pode ser representada por contiguidade ou por encadeamento. A primeira utiliza-se da definição de um vetor, tendo assim limite de tamanho e a segunda, conforme seu nome já diz, utiliza-se da definição de uma lista simplesmente encadeada para representá-la seguindo seus critérios (propriedade).

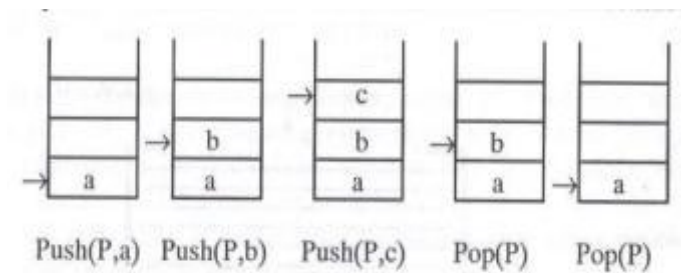
Apesar de uma pilha por encadeamento ser conceitualmente considerada infinita, na prática comprova o contrário, sendo ela dependente dos recursos de memória disponibilizados pelo computador.

Para assimilar os fundamentos de pilha nesta apostila, a ênfase será na pilha por contiguidade.

Manipulação de pilhas

Primeiro deve-se reservar um espaço na memória. O problema é que nem sempre é fácil determinar esse espaço. Se o espaço for muito pequeno, a pilha vai ultrapassar o espaço reservado, e se for muito grande, ficará como um espaço de memória desperdiçado.

Depois de reservado, deve-se escolher uma de suas extremidades para a base. Então se adicionam os elementos, em direção ao topo, formando assim, a pilha.



Como em uma pilha sempre há 'movimento', ou seja, a retirada e a inserção de elementos, devem mantê-la sempre atualizada. Em função disto, reserva-se uma posição na memória para guardar o endereço em que se encontra o topo da pilha, naquele momento. Essa posição de memória é denominada ponteiro para o topo da pilha.

Com a inserção de elementos, primeiro deve-se fazer com que o ponteiro aponte para o espaço vazio que se encontra no topo, então se insere o novo elemento.

Para eliminar um elemento da pilha, ajusta-se o ponteiro para o próximo elemento que então vai ficar no topo e desempilha o que antes estava no topo.

Operações em Pilha

As principais operações realizadas em uma pilha por contiguidade são:

- **Push/empilhar:** Adiciona um elemento ao topo da pilha.
- **Pop/desempilhar:** Remove o elemento do topo da pilha.
- **Top/topo:** Retorna o elemento do topo da pilha sem removê-lo.
- **Size/tamanho:** Retorna o número de elementos na pilha.
- **Is Empty/eVazia:** Verifica se a pilha está vazia.

Vantagens

- **Implementação simples:** A pilha por contiguidade é fácil de implementar e entender.
- **Acesso rápido aos elementos:** O acesso aos elementos da pilha é feito em tempo constante $O(1)$.
- **Utiliza memória de forma eficiente:** A pilha por contiguidade utiliza memória de forma eficiente, pois os elementos são armazenados em um vetor contíguo.

Desvantagens

- **Tamanho fixo:** O tamanho da pilha é fixo e deve ser especificado no momento da criação. Se a pilha ficar cheia, é necessário realocar o vetor para um tamanho maior.
- **Inserção e remoção no meio da pilha:** A inserção e remoção de elementos no meio da pilha são operações caras, pois envolvem o deslocamento de todos os elementos acima do elemento inserido ou removido.

Aplicações Práticas

- Controle de entrada e saída de sub-rotinas

Uma chamada de função (normalmente em qualquer linguagem) usa um recurso chamado de pilha. A pilha é usada para armazenar os endereços de retorno das funções bem como os seus parâmetros. A pilha de endereços funciona como uma pilha de papeis. Cada vez que uma nova função é executada, o endereço de retorno (aquele no qual o programa deve retomar a execução após a chamada) é guardado na pilha. A função é então executada. Ao seu término, o endereço de retorno é retirado da pilha e a execução continua a partir deste endereço. O funcionamento da pilha é muito importante para entender como funcionam as chamadas recursivas.

- Avaliação de expressões aritméticas

As pilhas podem ser empregadas para a avaliação de expressões aritméticas, que, por si só, é um importante tópico da Ciência da Computação.

As operações são efetuadas de acordo com a ordem determinada pelas regras usuais da aritmética: multiplicação e divisão têm prioridade sobre adição e subtração, sendo que operadores de mesma precedência são executados na ordem em que aparecem, da esquerda para direita. Os parênteses são usados para alterar a ordem natural de precedência, determinando uma precedência maior.

O fato de parêntese alterar a precedência e de haver naturalmente precedência entre operadores dificultam a avaliação de expressões. Felizmente, um lógico polonês conseguiu criar uma representação para expressões onde não existem prioridades e nem necessidade de parênteses, que é a Notação Polonesa Reversa.

O algoritmo desta notação implica numa disciplina onde o último operador encontrado é o primeiro a ser copiado na saída. Logo, uma pilha é perfeitamente cabível para esta aplicação. Por exemplo, uma operação do tipo $A+B*C$ é transformada para $ABC *+$, e a pilha poderá implementá-la de forma a fazer a avaliação naturalmente.

- Conversão de Base

A conversão de base é um exemplo de característica de pilha. Se você não sabe, para convertermos um número de uma base para outra base, devemos dividi-lo sucessivamente pela base desejada até atingir o quociente igual a 0. Neste momento, os restos obtidos nas divisões devem ser tomados em ordem inversa.

- Gerenciamento de memória, etc.

As pilhas por contiguidade são uma estrutura de dados simples e eficiente que pode ser usada em uma variedade de aplicações. No entanto, é importante ter em mente as limitações da pilha por contiguidade, como o tamanho fixo e a dificuldade de inserção e remoção no meio da pilha.

7.1 OPERAÇÕES BÁSICAS DA PILHA POR CONTIGUIDADE

Classe da declaração da pilha:

```
public class PilhaContig {
    private Item [] info;
    private int topo;

    public PilhaContig(int qte){
        this.topo = 0;
        this.info = new Item [qte];
    }
    public Item getInfo(){
        return this.info[this.topo-1];
    }
    public int getTopo(){
        return this.topo;
    }
    public boolean eVazia(){
        return (this.topo == 0);
    }
    public boolean eCheia(){
        return (this.topo == this.info.length);
    }
    //inserir um novo dado no topo da pilha.
    public boolean empilhar (Item elem){
        if (this.eCheia())
            return false;
        else {
```

```
        this.info[this.topo]= elem;
        this.topo++;
        return true;
    }
}
//remove o dado que está no topo da pilha (somente um dado).
public Item desempilhar(){
    if (this.eVazia())
        return null;
    else{
        this.topo--;
        return this.info[this.topo];
    }
}
}
```

Aplicação para manipular uma pilha por contiguidade:

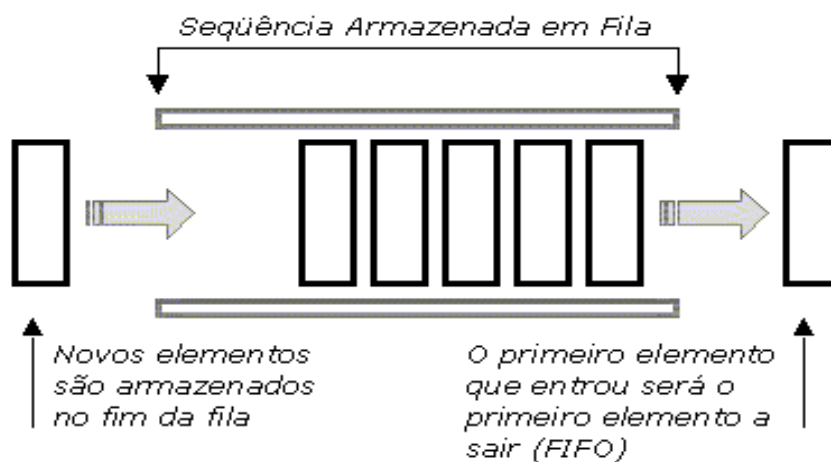
```
package pilhaContig;
import java.util.Scanner;
import dados.Item;
public class MenuPrincipal {
    static Scanner scan = new Scanner(System.in);
    public static void main(String[] args) {
        System.out.println ("digite o tamanho máximo da pilha");
        int quant = scan.nextInt();
        //instancia uma nova pilha.
        PilhaContig pilha = new PilhaContig(quant);
        int valor;
        Item item;
        char opcao;
        do {
            opcao = menu();
            switch (opcao){
                case '1':
                    System.out.println ("Digite um valor para inserir na "+
                        +"pilha");
                    valor = scan.nextInt();
                    if (! pilha.empilhar(new Item (valor)))
                        System.out.println("Pilha está cheia");
                    break;
                case '2':
                    item = pilha.desempilhar();
                    if (item == null)
                        System.out.println("A pilha está vazia");
                    else
                        System.out.println(item.getChave());
                    break;
                case '3':
                    System.out.println("fim do programa");
                    break;
                default:
                    System.out.println("opção inválida, tente novamente");
            }
        } while (opcao!='3');
        System.exit(0);
    }
}
```

```
    public static char menu(){
        System.out.println("Escolha uma Opção:\n" +
            "1. empilhar\n"+
            "2. desempilhar\n"+
            "3. Sair");
        return scan.next().charAt(0);
    }
}
```


Capítulo 8 - FILAS

Conceito

Uma fila é uma coleção de itens numa espécie de lista linear, em que as inserções são realizadas num extremo, que chamamos de Trás, ficando as remoções restritas ao outro, chamado de Frente. O extremo onde os elementos são inseridos é denominado 'final' da fila, e aquele de onde são removidos é denominado 'começo' da fila. Devido a esta disciplina de acesso, o primeiro elemento que entra é exatamente o primeiro que sai. Daí o fato de estas listas serem também denominadas listas **FIFO** (First-In/First-Out). O interessante desta estrutura, é que aqui, não existe uma preocupação com uma organização crescente dos elementos da fila, mas sim com a ordem de entrada e saída dos mesmos.



Um exemplo bastante comum de filas verifica-se num balcão de atendimento, onde pessoas formam uma fila para aguardar até serem atendidas. Naturalmente, devemos desconsiderar os casos de pessoas que "furam" a fila ou desistam de aguardar. O tipo abstrato que veremos não suporta inserções nem remoções no meio da lista.

A palavra **queue**, da língua inglesa, significa fila. Algumas vezes, você encontrará este termo, em vez de fila.

Por definição, uma lista FIFO é uma estrutura dinâmica, ou seja, é uma coleção que pode aumentar ou diminuir durante sua existência. Aliás, o nome fila, usado como sinônimo para FIFO, advém justamente da semelhança entre o modo como as listas FIFO crescem e/ou diminuem e o modo como as filas, no mundo real, funcionam.

Deve ficar bem claro que nesta estrutura os elementos são colocados no final da fila e que só temos acesso ao primeiro elemento da mesma. Importante: não é possível percorrer os elementos da fila sem destruí-la.

Propriedade:

FIFO ("First-in, First-out") = PRIMEIRO a entrar, PRIMEIRO a sair.

Sistemas operacionais utilizam filas para regular a ordens na qual tarefas devem receber processamento e recursos devem ser alocados a processos.

Outro exemplo de utilização de filas em computação é a implementação de uma fila de impressão. Se uma impressora é compartilhada por várias máquinas, deve-se adotar uma estratégia para determinar que documento será impresso primeiro. A estratégia mais simples é tratar todas as

requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos – o primeiro submetido é o primeiro a ser impresso.

Representação

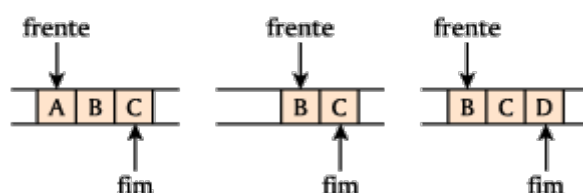
As filas podem ser implementadas de duas formas: sequencial ou encadeada. Para sequencial, implica alocação com tamanho fixo, e para encadeada, alocação dinâmica utilizando a representação de uma lista simplesmente encadeada com as propriedades de uma fila.

Para assimilar os fundamentos de fila nesta apostila, a ênfase será na fila por contiguidade.

Manipulação de filas

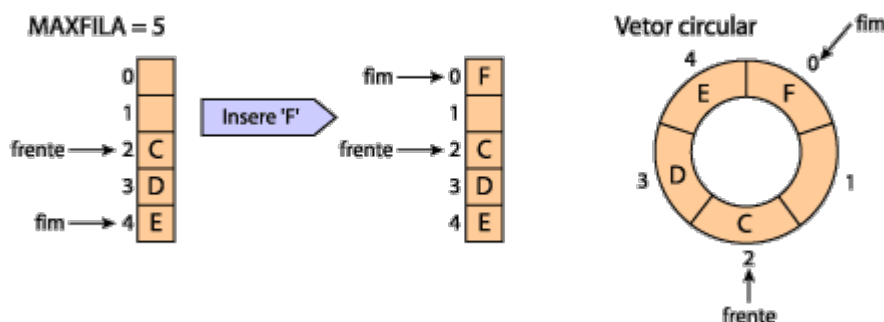
A implementação pode ser feita através de um bloco de posições de memória consecutivas, semelhante ao usado para armazenar uma pilha. Porém, agora se reserva duas posições de memória para os apontadores, um para indicar o início da fila e outro para indicar o final da fila.

Primeiro origina-se uma fila vazia, com os dois ponteiros apontando para a mesma posição. Assim que um elemento é inserido, ele vai ocupar a posição apontada pelo ponteiro para o final da fila, então, este é ajustado para apontar para a próxima posição livre. Assim, o ponteiro para o final da fila sempre apontará para o primeiro espaço vazio após o final da fila. E o ponteiro para o início da fila sempre apontará para o primeiro elemento da fila.



Um problema com esse tipo de estrutura é semelhante ao que acontece na vida real. Uma fila pequena, mas ativa, pode solicitar mais recursos de memória do que uma fila grande, mas inativa, ou seja, o problema não está no tamanho da fila, mas no procedimento de acesso. Uma solução para esse suposto problema é deslocar os elementos para frente da fila, como acontece na vida real: numa fila de banco, quando uma pessoa vai ser atendida, as pessoas que ainda estão na fila dão um passo à frente.

Uma solução é denominada fila circular, que seria reservar um espaço na memória para a fila e começá-la em uma das extremidades do bloco de memória e deixar que a fila se estenda até a outra extremidade do bloco, assim, os espaços que estão à frente vão esvaziando. Quando os espaços do fim da fila já estiverem vazios, e novos elementos forem incrementados, ocuparão os espaços do início da fila. Assim, a fila circula em torno de si mesma, em vez de se expandir pelo bloco de memória.



8.1 FILA CIRCULAR

Essa técnica resulta na implementação denominada fila circular, pois o processo se baseia na utilização cíclica do bloco de posições de memória reservado para a fila. No que se refere à fila, a última posição do bloco reservado à fila é considerada adjacente à sua primeira posição.

Devemos observar a diferença entre a estrutura conceitual visualizada pelo usuário de uma fila e a estrutura cíclica real implementada na memória da máquina. Essas diferenças são superadas com o auxílio do software: juntamente com o conjunto de posições de memória utilizado para o armazenamento de dados, a implementação de fila inclui um conjunto de procedimentos que interpretam os dados armazenados de acordo com as normas que regem o funcionamento de uma fila. Tais procedimentos contêm rotinas para inserir e remover elementos da fila, bem como para determinar se a fila está vazia ou lotada. Por meio delas, um programador pode solicitar a inserção ou remoção de elementos utilizando para tais operações rotinas pré-elaboradas de software, sem se deter nos detalhes da real maneira como é efetuado seu armazenamento em memória.

Aplicações Práticas

- Gerência de dados/ processos por ordem cronológica:
 - fila de impressão em uma impressora de rede;
 - fila de pedidos de uma expedição ou tele-entrega;
- Simulação de processos sequenciais:
 - chão de fábrica: fila de camisetas a serem estampadas;
 - Comércio: simulação de fluxo de um caixa de supermercado;
 - Tráfego: simulação de um cruzamento com um semáforo.

8.2 OPERAÇÕES BÁSICAS DA FILA CIRCULAR

Classe da declaração da fila:

```
package fila;
import dados.Item;

public class FilaCircular {
    private Item [] info;
    private int frente;
    private int tras;
    private int tamanho;

    public FilaCircular(int qte){
        this.frente = 0;
        this.tras = 0;
        this.tamanho = 0;
        this.info = new Item [qte];
    }
    public Item getInfo(){
        return this.info[this.frente];
    }
    public int getFrente(){
        return this.frente;
    }
    public int getTras(){
        return this.tras;
    }
    public int getTamanho(){
        return this.tamanho;
    }
    public boolean eVazia(){
```

```
        return (this.tamanho == 0);
    }
    public boolean eCheia(){
        return (this.tamanho == this.info.length);
    }
    public boolean enfileirar (Item elem){
        if (this.eCheia())
            return false;
        else {
            this.info[this.tras]= elem;
            this.tras = (++this.tras % this.info.length);
            this.tamanho++;
            return true;
        }
    }
    public Item desenfileirar(){
        Item no;
        if (this.eVazia())
            return null;
        else{
            no = this.info[this.frente];
            this.frente = (++this.frente % this.info.length);
            this.tamanho--;
            return no;
        }
    }
    public String toString(){ //imprimir o conteúdo da fila
        String msg="";
        int aux= this.frente;
        for (int i=1; i <= this.tamanho; i++){
            msg+= this.info[aux].getChave()+" ";
            aux= (++aux % this.info.length);
        }
        return msg;
    }
}
```

Aplicação para manipular a uma fila circular:

```
package fila;
import java.util.Scanner;
import dados.Item;

public class BlocoPrincipalFilaCircular {
    static Scanner scan = new Scanner(System.in);
    public static void main(String[] args) {
        System.out.println("digite o tamanho máximo da fila");
        int tam = scan.nextInt();
        FilaCircular fila = new FilaCircular(tam);
        int valor;
        Item item;
        char opcao;
        do {
            System.out.println("Escolha uma Opção:\n"+
                                "1. enfileirar\n"+
                                "2. desenfileirar\n"+
                                "3. Imprimir Fila\n"+
                                "4. Sair");
            opcao = scan.next().charAt(0);
            switch (opcao){
                case '1':
                    System.out.println("Inserir um Valor no final da fila\n");
```

```
        + "Digite um valor");
    valor = scan.nextInt();
    if (! fila.enfileirar(new Item (valor))) {
        System.out.println("fila está cheia");
    }
    break;
case '2':
    item = fila.desenfileirar();
    if (item == null) {
        System.out.println("A fila está vazia");
    } else {
        System.out.println("o " + item.getChave() +
            " foi removido da fila");
    }
    break;
case '3':
    if (!fila.eVazia()) {
        System.out.println("A fila: " + fila.toString());
    } else {
        System.out.println("A fila está vazia");
    }
    break;
case '4':
    System.out.println("fim do programa");
    break;
default:
    System.out.println("opção invalida, tente novamente");
}
} while (opcao != '4');
System.exit(0);
}
}
```

Capítulo 9 - RECURSIVIDADE

Um algoritmo que, para resolver um problema, divide-o em subproblemas mais simples, cujas soluções requerem a aplicação dele mesmo, é chamado RECURSIVO.

Em termos de programação, UMA ROTINA É RECURSIVA quando ela chama a si mesma, seja de forma direta ou indireta.

Os processos recursivos são tratados internamente pela máquina principalmente através do uso de mecanismos de empilhamento, cada vez que um subprograma chama a si próprio, mais informações são colocadas na pilha sendo retiradas à medida que os resultados vão sendo obtidos e assim sucessivamente. O número de chamadas recursivas é dito profundidade de recorrência.

A recursividade é um conceito fundamental na ciência da computação, onde uma função chama a si mesma para resolver um problema. Essa técnica elegante permite decompor problemas complexos em instâncias menores e mais simples, facilitando a criação de algoritmos concisos e eficientes.

O Princípio da Divisão e Conquista

A recursividade se baseia no princípio da divisão e conquista, onde um problema é dividido em subproblemas menores, que são resolvidos recursivamente. As soluções dos subproblemas são então combinadas para formar a solução do problema original.

Elementos Essenciais da Recursividade

Uma função recursiva possui dois elementos essenciais:

1. **Caso base (solução trivial):** Uma condição de parada que impede a função de se chamar infinitamente.
2. **Caso recursivo (solução geral):** A chamada recursiva da função, com um ou mais parâmetros modificados para se aproximarem do caso base.

SOLUÇÃO TRIVIAL:

0! = 1 {dada por definição}

SOLUÇÃO GERAL:

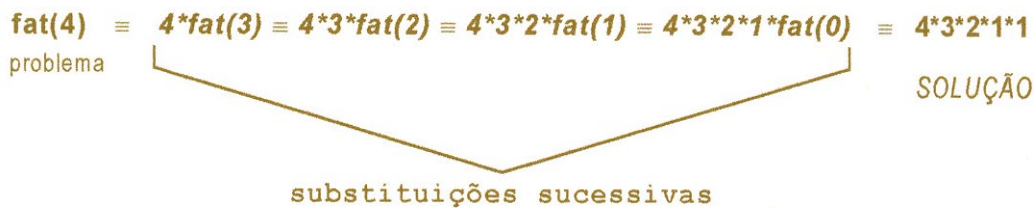
n! = n*(n-1)! {requer reaplicação da rotina para (n-1)!}

Considerando **f(n) = n**, então **n=0** implica numa condição de parada do mecanismo recursivo, garantindo o término do algoritmo que calcula o fatorial:

```
public static void main(String [] args){
    int num = 5;
    System.out.println("o fatorial do número "+num+ " é "+fat(num));
}

public static int fat (int n) {
    if (n == 0)
        return 1;
    else
        return (n * fat (n-1));
}
```

Em termos matemáticos, a recursão é uma técnica que, através de substituições sucessivas, reduz o problema a ser resolvido a um caso de solução trivial. Por exemplo, o cálculo de **fat(4)** na figura abaixo:



A versão não recursiva é assim:

```
public static int fat_iterativo (int n){
    int f = 1;
    for (int i = 1; i <= n; i++) {
        f *= i;
    }
    return f;
}
```

Melhores Práticas para o Uso da Recursividade

- **Definir um caso base claro:** O caso base é crucial para evitar recursão infinita.
- **Garantir que o caso recursivo se aproxime do caso base:** A cada chamada recursiva, os parâmetros devem se aproximar da condição de parada.
- **Evitar recursão excessiva:** A recursão pode consumir muita memória e tempo de processamento. Em alguns casos, a iteração pode ser uma alternativa mais eficiente.
- **Utilizar recursão de cauda quando possível:** A recursão de cauda é uma forma de recursão que pode ser otimizada pelo compilador, reduzindo o consumo de memória.

Quando Usar Recursividade

A recursividade é uma técnica poderosa para resolver problemas que podem ser decompostos em subproblemas semelhantes. No entanto, é importante avaliar se a recursividade é a melhor abordagem para cada caso, considerando os custos de memória e desempenho.

Outros exemplos:

1) Exemplo de um programa que imprime na tela a soma dos n primeiros números inteiros positivos.

```
public class somarInteiros {
    public static void main(String[] args) {
        int num=InOut.leInt("digite um número inteiro:");
        System.out.println("", "a soma é "+somarInt(num));
        System.exit(0);
    }
    static int somarInt(int n){
        if (n==0)
            return 0;
        else
            return (somarInt(n-1)+n);
    }
}
```

2) Exemplo de um programa que calcula a potencia de um número.

```
public class calcularPotencia {
    public static void main(String[] args) {
        int base=InOut.leInt("digite um número inteiro para base:");
        int exp=InOut.leInt("digite um número inteiro para o expoente:");
        System.out.println("", "a potencia de "+base+" elevado a "+exp+" é "+potencia(base,exp));
        System.exit(0);
    }
}
```

```
    }  
    static int potencia(int b,int e){  
        if (e==1)  
            return b;  
        else  
            return (potencia(b,e-1)*b);  
    }  
}
```

3) Exemplo de um programa para verificar se um caracter específico está ou não em uma string.

```
public class caracterString {  
    public static void main(String[] args) {  
        String palavra= InOut.leString("Digite uma palavra:");  
        char letra= InOut.leChar("Digite um caracter:");  
        int i=0;  
        if (esta(palavra,letra,i))  
            System.out.println ("", "o caracter está na palavra");  
        else  
            System.out.println("", "o caracter não está na palavra");  
        System.exit(0);  
    }  
    static boolean esta(String p, char le, int pos){  
        if (pos == p.length())  
            return false;  
        else  
            if (p.charAt(pos)== le)  
                return true;  
            else  
                return esta(p, le, pos+1);  
    }  
}
```

Aplicação da recursão

Embora a recursão seja uma ferramenta bastante interessante para resolução de problemas, nem sempre ela poderá ser empregada. Enquanto alguns problemas têm solução imediata com o uso de recursão, outros são praticamente impossíveis de se resolver de forma recursiva. É preciso analisar o problema e verificar se realmente vale a pena tentar encontrar uma solução recursiva.

A recursão, se bem utilizada, pode tornar um algoritmo muito elegante; isto é, claro, simples e conciso. Porém, na maioria das vezes, uma solução iterativa (não recursiva) será mais eficiente. Cada chamada recursiva implica em um custo tanto de tempo quanto de espaço, pois cada vez que a rotina é chamada, todas as variáveis locais são recriadas.

Apesar de suas aparentes vantagens como a clareza de interpretação do código ou a elegância na implementação, os programas recursivos apresentam também alguns perigos como por exemplo à dificuldade na localização de erros e em alguns casos a falta de eficiência devida principalmente à questão de espaço, pois como já foi dito anteriormente as informações são armazenadas em uma pilha e depois retiradas. Portanto essa quantidade de informação pode ser proporcional à profundidade de recorrência e de tempo gasto para alocar memória, copiar informações etc. Outro ponto a ser observado diz respeito ao critério de parada responsável pela limitação das chamadas recursivas que se não for devidamente definido pode levar ao consumo de toda a memória disponível.

Assim, determinar precisamente se devemos usar recursão ou não, ao resolver um problema, é uma questão bastante difícil de ser respondida. Pois, para isto, teríamos que comparar as possíveis soluções recursiva e iterativa.

Capítulo 10 - LISTA NÃO LINEAR - ÁRVORE

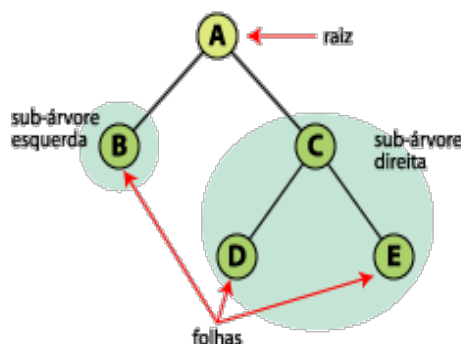
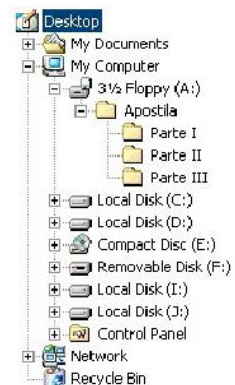
Conceito

Árvore é um tipo de estrutura de dados semelhante à Lista Duplamente Encadeada, onde cada elemento tem dois ou mais sucessores e todos eles possuem um antecessor. Esta é uma estrutura não linear. Quando em uma árvore todos os pais não têm mais de um filho, então podemos considerar como uma lista.

O nome árvore é dado por uma analogia às árvores vegetais onde o ponto de partida é a raiz e o ponto mais extremo é a folha, assim como na árvore como estrutura de dados. Qualquer elemento é chamando de nó. O nó que contém o maior número de ramificações contém o grau da árvore.

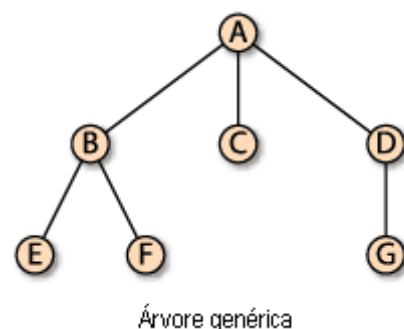
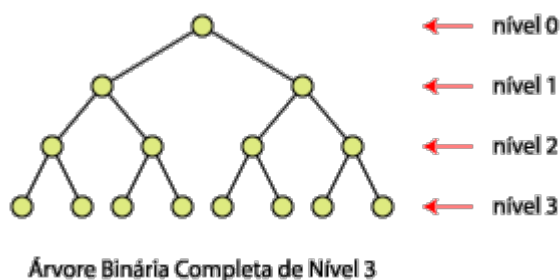
As árvores são estruturas de dados usadas para representar hierarquias. Um exemplo de árvore são as árvores genealógicas. Outro exemplo de árvore é a árvore de diretório do computador.

Uma árvore é composta por um conjunto de nós. Existe um nó denominado *raiz*, que contém nenhuma ou mais subárvores, cujas raízes estão diretamente ligadas à raiz (principal). Esses nós das raízes das subárvores são denominados *filhos* do nó *pai* (raiz). Os nós que não têm filhos são chamados de *folhas*.



O grau do nó de uma árvore corresponde ao número de filhos, e a profundidade de uma árvore corresponde ao número de nós compreendidos na trajetória da raiz até uma folha, ou seja, é o número de camadas horizontais de uma árvore.

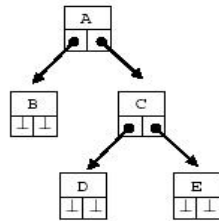
Existem vários tipos de árvores, as principais são as genéricas e as binárias. As árvores genéricas são aquelas que possuem um nó-raiz e outros subconjuntos. As árvores binárias são aquelas que possuem um nó-raiz e no máximo duas subárvores, chamadas de subárvore esquerda e direita. O grau de uma árvore binária é limitado a dois.



Implementação de árvores

As árvores possuem uma estrutura semelhante à das listas ligadas. Cada nó contém três componentes: o dado e dois ponteiros: um para cada filho (direito e esquerdo). Cada nó da árvore é representado por um pequeno bloco de posições de memória, de acordo com a estrutura desejada da árvore.

Árvores binárias – representação

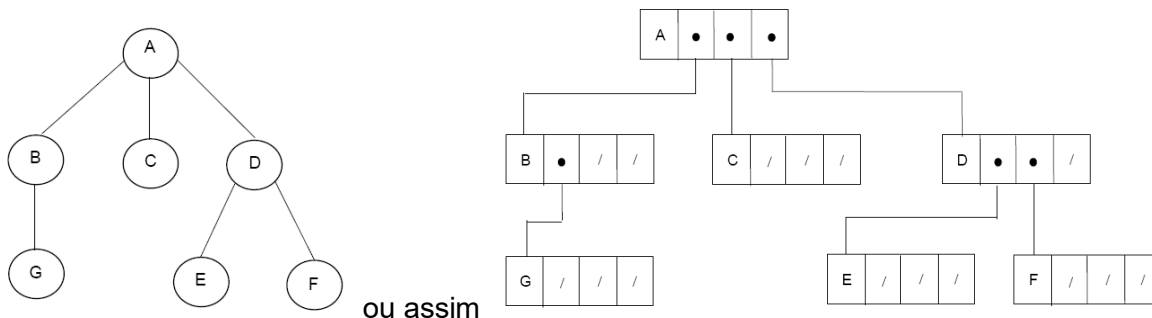


Cada ponteiro apontará para o filho direito ou esquerdo do nó correspondente, ou então é associado ao valor NULL, caso não haja mais nós nesta direção da árvore. Também se reserva uma posição especial na memória estática para armazenar o endereço do nó raiz, denominado ponteiro para a raiz da árvore.

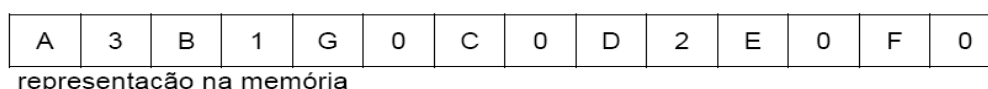
Existem duas formas de implementar árvore:

- ♦ **Por Encadeamento** – é a forma mais adequada, permitindo, com igual facilidade, manipulações das árvores, bem como diversas ordens de acesso aos nós. cada nó é um dado que pode ser alocado dinamicamente e possui espaço para representar tanto a informação do nó como as referências das subárvores daquele nó;

Exemplo gráfico:



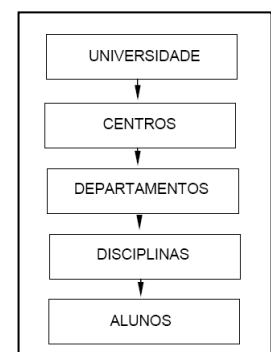
- ♦ **Por contiguidade ou Alocação por Adjacência** – utiliza-se quando os nós da árvore devem ser processados na mesma ordem em que aparecem na alocação sequencial. Ocorre quando a árvore está em armazenamento permanente (em fita ou disco magnético) de árvores representadas por encadeamento. Exemplo gráfico da árvore anterior:



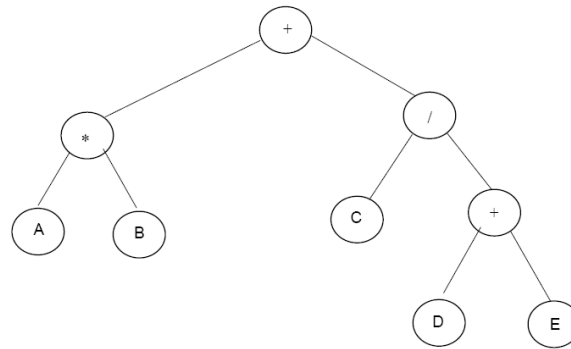
Portanto, a alocação sequencial não constitui na maioria dos casos uma maneira conveniente para representar árvores (dificuldades para inserções, remoções e localizações de um nó particular. Quando for necessário efetuar manipulações sobre a árvore, ela é transformada para a forma encadeada e, então, manipulada.

Aplicações Práticas

- Administração de Grandes Quantidades de Dados em uma Ordem qualquer. Árvores permitem o uso de listas encadeadas e a divisão de um conjunto de dados em subconjuntos;
- Administração de Informações Hierárquicas. Posso representar que conjuntos de dados dependem de outros de uma forma natural. Exemplo hierárquico de uma universidade:
- Representação de Processos Decisórios;
- Avaliação de expressões aritméticas;



- Equação $\rightarrow A * B + C / (D + E)$



- Compiladores;
- Pesquisa de dados;
- Classificação de dados; etc.

Uma árvore possui as seguintes características:

- A linha que liga dois nodos da árvore denomina-se aresta.
- Nó raiz – nó do topo da árvore, do qual descendem os demais nós. É o primeiro nó da árvore;
- Nó interior – nó do interior da árvore (que possui descendentes);
- Nó terminal ou Folha – nó que não possui descendentes; qualquer nó com grau zero;
- Trajetória ou Nível – número de nós que devem ser percorridos até o nó determinado;
- Grau do nó – número de nós descendentes do nó, ou seja, o número de subárvores de um nó;
- Grau da árvore – é o número máximo de subárvores de um nó;
- Altura da árvore – número máximo de níveis dos seus nós;
- Altura do nó – número máximo de níveis dos seus nós.

Outras definições:

- ÁRVORE COMPLETA se todos os nós têm grau M, com exceção dos nós do último nível, que têm grau zero.
- ÁRVORE BALANCEADA se as alturas das subárvores esquerda e direita diferem de, no máximo, uma unidade.

10.1 ÁRVORE BINÁRIA

Em uma árvore binária cada nó tem no máximo duas subárvores, e quando há somente uma é necessário distinguir entre subárvore esquerda e direita. Árvores binárias podem ser vistas em diversas situações do cotidiano. Por exemplo, um torneio de futebol eliminatório, do tipo das copas dos países, como a Copa do Mundo, em que a cada etapa os times são agrupados dois a dois e sempre são eliminados metade dos times é uma árvore binária.

Formalmente uma árvore binária pode ser definida como um conjunto finito de nós, que é vazio, ou consiste de um nó raiz e dois conjuntos disjuntos de nós, a subárvore esquerda e a subárvore direita. É importante observar que uma árvore binária não é um caso especial de árvore e sim um conceito completamente diferente.

Como já foi comentado, uma árvore pode ser entendida como uma lista em que cada elemento tem apenas um antecessor e, como estamos estudando as árvores binárias, tem no máximo dois sucessores no caso. Cada nó dessa árvore, portanto, é um endereço na memória que contém espaço para os dados que serão armazenados naquele endereço e ainda contém, no caso das árvores binárias, dois ponteiros que, embora não existam lados direito nem esquerdo na memória de uma máquina, consideramos que apontam um para o filho mais à esquerda do nó e o outro

aponta para o filho mais à direita.

As folhas, que não têm filhos, têm os ponteiros aterrados, para que não apontem para nenhum endereço de memória e ainda pode ser criado um apontador que vai indicar onde está a raiz da árvore.

As operações básicas de um objeto do tipo árvore são criar uma árvore vazia, inserir um item na árvore, retirar um item da árvore, procurar por um item na árvore e mostrar todo o conteúdo de uma árvore. As árvores podem ser aplicadas das seguintes maneiras: administrar uma grande quantidade de dados em qualquer ordem, administrar dados hierárquicos, em organogramas, em diretórios, etc.

10.2 ÁRVORES DE PESQUISA

A Árvore de Pesquisa é uma estrutura de dados muito eficiente para armazenar informação. Ela é particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de requisitos tais como:

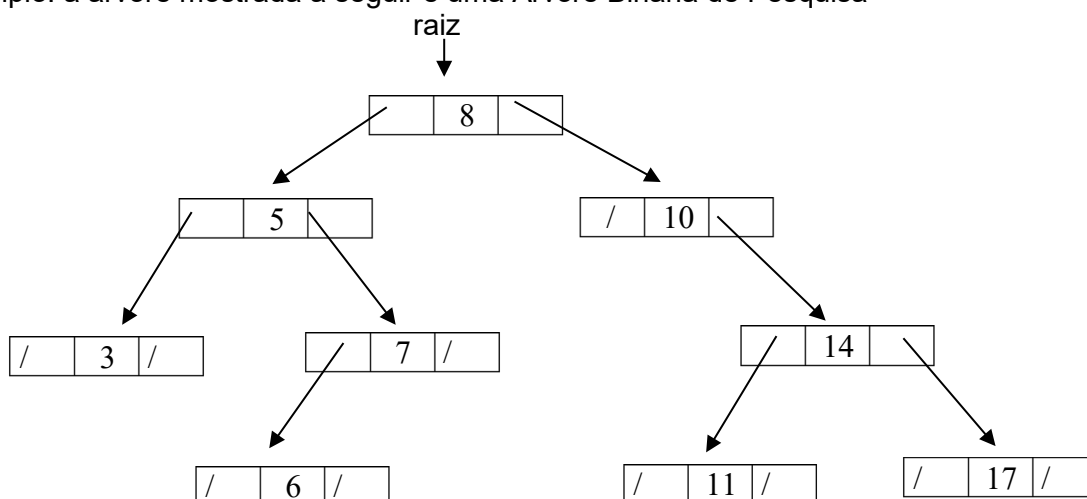
- Acesso direto e sequencial eficientes;
- Facilidade de inserção e remoção de registros
- Alta taxa de utilização de memória;
- Utilização de memória primária e secundária.

10.3 ÁRVORE BINÁRIA DE PESQUISA

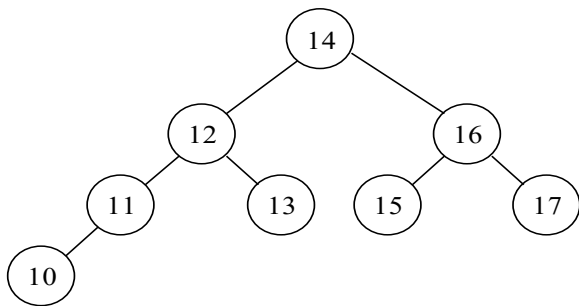
Uma Árvore Binária de Pesquisa (ABP) é uma Árvore Binária em que todo nó contém um registro e para cada nó a seguinte propriedade é verdadeira:

Todos os registros com chaves menores do que a Raiz estão na sub-árvore esquerda e todos os registros com chaves maiores do que a Raiz estão na sub-árvore direita.

Exemplo: a árvore mostrada a seguir é uma Árvore Binária de Pesquisa



Outra forma de representação gráfica de uma Árvore está a seguir após a inserção dos nós com os números: 14;16;12;11;17;15;10;13;



As árvores binárias de pesquisa (ABPs) são usadas em diversas aplicações práticas, especialmente quando a organização e a busca eficiente de dados são cruciais. Aqui estão alguns exemplos:

1. Bancos de dados e sistemas de busca:

- **Indexação:** ABPs são usadas para indexar registros em bancos de dados, permitindo buscas rápidas por chaves específicas. Por exemplo, em um banco de dados de clientes, uma ABP pode indexar os registros pelo nome do cliente, permitindo que o sistema encontre rapidamente os dados de um cliente específico.
- **Mecanismos de busca:** Embora mecanismos de busca modernos usem estruturas mais complexas, os princípios das ABPs são aplicados em algumas partes do processo, como na organização de palavras-chave e na busca de termos em índices.

2. Compiladores e interpretadores:

- **Tabelas de símbolos:** Compiladores e interpretadores usam ABPs para armazenar e buscar informações sobre variáveis, funções e outros símbolos em um programa. Isso permite que o compilador verifique rapidamente se um símbolo foi declarado e acesse suas propriedades.

3. Sistemas de arquivos:

- **Organização de diretórios:** Em alguns sistemas de arquivos, ABPs podem ser usadas para organizar os diretórios e arquivos em uma hierarquia. Isso facilita a busca de arquivos por nome e a navegação na estrutura de diretórios.

4. Aplicações de dicionário e agenda:

- **Dicionários eletrônicos:** ABPs podem ser usadas para armazenar palavras e suas definições, permitindo buscas rápidas por palavras.
- **Agendas eletrônicas:** Em agendas eletrônicas, ABPs podem ser usadas para armazenar compromissos e contatos, permitindo buscas rápidas por data, nome ou outros critérios.

5. Roteadores de rede:

- **Tabelas de roteamento:** Roteadores de rede usam ABPs para armazenar e buscar informações sobre rotas de rede. Isso permite que o roteador encaminhe pacotes de dados para o destino correto de forma eficiente.

É importante notar que, em muitas aplicações modernas, variantes balanceadas de ABPs, como árvores AVL e árvores rubro-negras, são preferidas para garantir um desempenho consistente em todos os casos.

A escolha de uma ABP ou outra estrutura de dados depende dos requisitos específicos da aplicação, como o tamanho do conjunto de dados, a frequência das operações de busca e a necessidade de balanceamento.

CAMINHAMENTO em Árvore Binária

Consiste em percorrer todos os nós da árvore, com o objetivo de consultar e/ou alterar a informação neles contida.

Existem vários métodos de caminhamento em árvore, que permitem percorrê-la de forma sistemática e de tal modo que cada nó seja “visitado” apenas uma única vez.

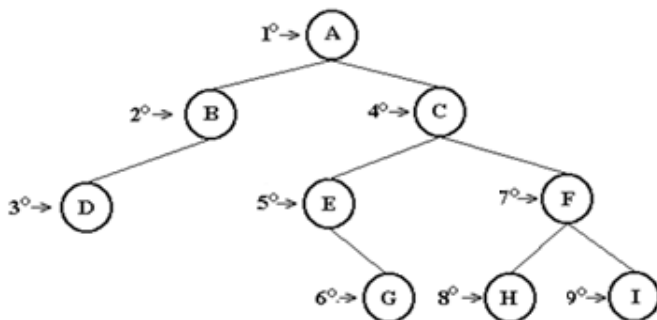
Um caminhamento completo sobre uma árvore produz uma sequência linear dos nós, de maneira que cada nó da árvore passa a ter um “seguinte” ou um nó “anterior”, ou ambos, para uma dada forma de caminhamento.

Há três maneiras recursivas de se percorrer árvores binárias, veja a seguir:

a) **Caminhamento Pré-fixado ou Travessia em Pré-ordem**

- 1 – visita a raiz
- 2 – caminha na subárvore esquerda
- 3 – caminha na subárvore direita

Exemplo: Considere a seguinte árvore



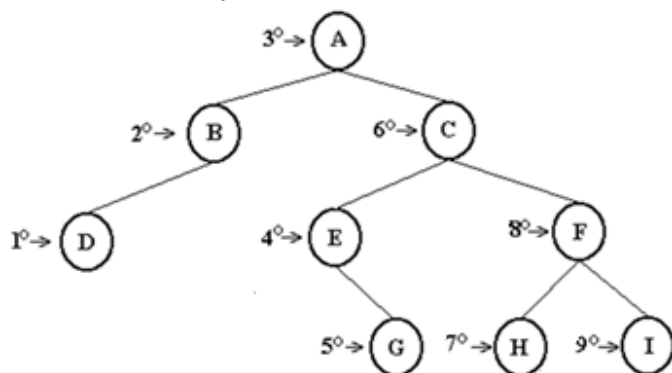
A sequência gerada pelo Caminhamento PRÉ-FIXADO é: { ABDCEGFHI }

Os algoritmos não recursivos utilizam uma pilha que armazena os endereços dos nós percorridos, isto é, “marca” o caminho.

b) **Caminhamento Central ou Travessia em In-Ordem**

- 1 – caminha na subárvore esquerda
- 2 – visita a raiz
- 3 – caminha na subárvore direita

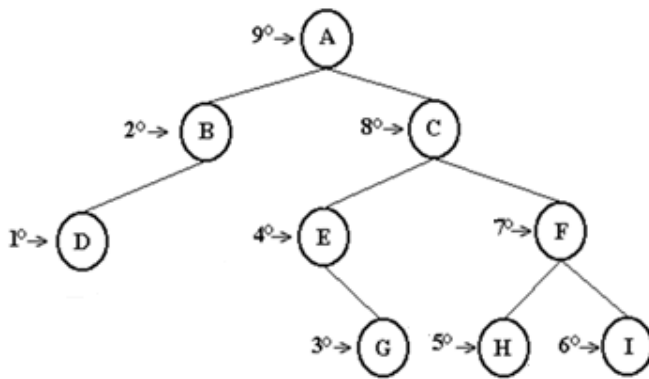
Para a mesma árvore do exemplo anterior, a sequência gerada pelo Caminhamento CENTRAL é: { DBAEGCHFI }



c) **Caminhamento PÓS-FIXADO ou Travessia em Pós-Ordem**

- 1 – caminha na subárvore esquerda
- 2 – caminha na subárvore direita
- 3 – visita a raiz

Para a mesma árvore do exemplo anterior, a sequência gerada pelo Caminhamento PÓS-FIXADO é: { DBGEHIFCA }



Remoção de um Nó

A operação de REMOÇÃO apresenta maiores dificuldades, pois existem 3 casos distintos que devem ser considerados:

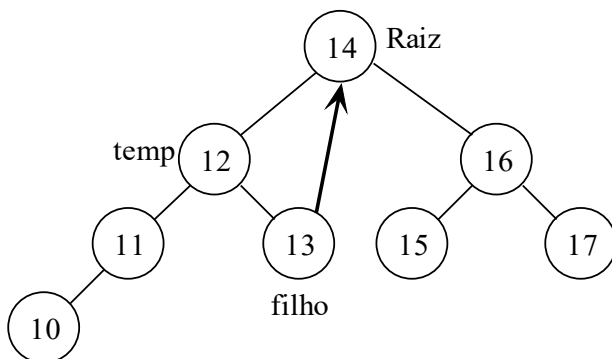
- remoção de nó folha;
- remoção de nó com um único filho
- remoção de nó com dois filhos

Os dois primeiros casos são simples, no entanto, o terceiro caso envolve um pouco mais de atenção. A ideia para resolver o caso **c)** é:

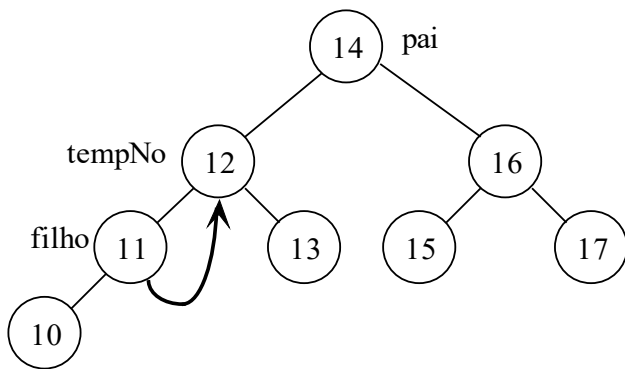
Trocar o nó removido pelo nó mais à direita na subárvore esquerda, ou pelo nó mais à esquerda na subárvore direita.

Ou seja, buscar o nó com a chave de maior valor da subárvore esquerda ou o nó com a menor chave da subárvore direita. Após a substituição no nó que desejava ser removido, estaremos com o caso **a)** ou com o caso **b)**.

Exclusão do elemento 14 da árvore:



Exclusão do elemento 12 da árvore:



10.4 OPERAÇÕES BÁSICAS DE ÁRVORE BINÁRIA DE PESQUISA

Classe da declaração do nó da árvore:

```
public class NoArv {
    private Item info; // o tipo Item está declarado no capítulo 1
    private NoArv esq, dir;

    public NoArv(Item elem){
        this.info = elem;
        this.esq = null;
        this.dir = null;
    }
    public NoArv getEsq(){
        return this.esq;
    }
    public NoArv getDir(){
        return this.dir;
    }
    public Item getInfo(){
        return this.info;
    }
    public void setEsq(NoArv no){
        this.esq = no;
    }
    public void setDir(NoArv no){
        this.dir = no;
    }
    public void setInfo(Item elem){
        this.info = elem;
    }
}
```

Classe da declaração da árvore binária de pesquisa:

```
public class Arvore {
    private NoArv raiz;
    private int quantNos; //opcional

    public Arvore(){
        this.quantNos=0;
        this.raiz = null;
    }
    public boolean eVazia (){
        return (this.raiz == null);
    }
    public NoArv getRaiz(){
        return this.raiz;
    }
    public int getQuantNos () {
```



```
        return this.quantNos;
    }
    //inserir um novo nó na árvore. Sempre insere em um atributo que seja igual a null
    public boolean inserir (Item elem){
        if (pesquisar (elem.getChave())){
            return false;
        }else{
            this.raiz = inserir (elem, this.raiz);
            this.quantNos++;
            return true;
        }
    }
    public NoArv inserir (Item elem, NoArv no){
        if (no == null){
            NoArv novo = new NoArv(elem);
            return novo;
        }else {
            if (elem.getChave() < no.getInfo().getChave()){
                no.setEsq(inserir(elem, no.getEsq()));
                return no;
            }else{
                no.setDir(inserir(elem, no.getDir()));
                return no;
            }
        }
    }
    //Pesquisa se um determinado valor está na árvore
    public boolean pesquisar (int chave){
        if (pesquisar (chave, this.raiz) != null){
            return true;
        }else{
            return false;
        }
    }
    private NoArv pesquisar (int chave, NoArv no){
        if (no != null){
            if (chave < no.getInfo().getChave()){
                no = pesquisar (chave, no.getEsq());
            }else{
                if (chave > no.getInfo().getChave()){
                    no = pesquisar (chave, no.getDir());
                }
            }
        }
        return no;
    }
    //remove um determinado nó procurando pela chave. O nó pode estar em qualquer
    //posição na árvore
    public boolean remover (int chave){
        if (pesquisar (chave, this.raiz) != null){
            this.raiz = remover (chave, this.raiz);
            this.quantNos--;
            return true;
        }
        else {
            return false;
        }
    }
    public NoArv remover (int chave, NoArv arv){
        if (chave < arv.getInfo().getChave()){
            arv.setEsq(remover (chave, arv.getEsq()));
        }else{
            if (chave > arv.getInfo().getChave()){
                arv.setDir(remover (chave, arv.getDir()));
            }else{
                if (arv.getDir() == null){
                    return arv.getEsq();
                }else{
                    if (arv.getEsq() == null){
                        return arv.getDir();
                    }
                }
            }
        }
    }
}
```

```
                }else{
                    arv.setEsq(Arrumar (arv, arv.getEsq()));
                }
            }
        }
    }
    return arv;
}
private NoArv Arrumar (NoArv arv, NoArv maior){
    if (maior.getDir() != null){
        maior.setDir(Arrumar (arv, maior.getDir()));
    }
    else{
        arv.setInfo(maior.getInfo());
        maior = maior.getEsq();
    }
    return maior;
}
//caminhamento central
public Item [] CamCentral (){
    int []n= new int[1];
    n[0]=0;
    Item [] vet = new Item[this.quantNos];
    return (FazCamCentral (this.raiz, vet, n));
}
private Item [] FazCamCentral (NoArv arv, Item [] vet, int []n){
    if (arv != null) {
        vet = FazCamCentral (arv.getEsq(),vet,n);
        vet[n[0]] = arv.getInfo();
        n[0]++;
        vet = FazCamCentral (arv.getDir(),vet,n);
    }
    return vet;
}
//caminhamento pré-fixado
public Item [] CamPreFixado (){
    int []n= new int[1];
    n[0]=0;
    Item [] vet = new Item[this.quantNos];
    return (FazCamPreFixado (this.raiz, vet, n));
}
private Item [] FazCamPreFixado (NoArv arv, Item [] vet, int []n){
    if (arv != null) {
        vet[n[0]] = arv.getInfo();
        n[0]++;
        vet = FazCamPreFixado (arv.getEsq(), vet,n);
        vet = FazCamPreFixado (arv.getDir(), vet,n);
    }
    return vet;
}
//caminhamento pós-fixado
public Item [] CamPosFixado (){
    int []n= new int[1];
    n[0]=0;
    Item [] vet = new Item[this.quantNos];
    return (FazCamPosFixado (this.raiz, vet, n));
}
private Item [] FazCamPosFixado (NoArv arv, Item[] vet, int []n){
    if (arv != null) {
        vet = FazCamPosFixado (arv.getEsq(), vet,n);
        vet = FazCamPosFixado (arv.getDir(), vet,n);
        vet[n[0]] = arv.getInfo();
        n[0]++;
    }
    return vet;
}
}
```

Aplicação para manipular árvore binária de pesquisa:

```
package arvore;
import java.util.Scanner;
import dados.Item;

public class BlocoPrincipalArvore {
    static Scanner scan = new Scanner(System.in);
    public static void main(String[] args) {
        Arvore arvore = new Arvore();
        int valor;
        Item [] vetor= new Item[10];
        char opcao;
        do {
            System.out.println("Escolha uma Opção:\n" +
                "1. Inserir Nó na árvore\n"+
                "2. Localizar Nó na árvore\n"+
                "3. Excluir Nó da árvore\n" +
                "4. Exibir árvore ordenada\n" +
                "5. Sair");
            opcao = scan.next().charAt(0);
            switch (opcao){
                case '1':
                    System.out.println("Digite um valor para inserir na árvore");
                    valor = scan.nextInt();
                    if (arvore.inserir(new Item(valor))){
                        System.out.println("inserção efetuada com sucesso");
                    }else{
                        System.out.println("valor já existe na árvore");
                    }
                    break;
                case '2':
                    if (arvore.eVazia()){
                        System.out.println("Árvore está vazia");
                    }else{
                        System.out.println("Digite o valor para pesquisar");
                        valor = scan.nextInt();
                        if (arvore.pesquisar(valor)){
                            System.out.println("valor+\" foi encontrado");
                        }else{
                            System.out.println("valor+\" não encontrado");
                        }
                    }
                    break;
                case '3':
                    if (arvore.eVazia()){
                        System.out.println("Arvore está vazia");
                    }else {
                        System.out.println("Digite um valor para excluir");
                        valor = scan.nextInt();
                        if (arvore.remover(valor)){
                            System.out.println("remoção efetuada");
                        }else{
                            System.out.println("valor não encontrado");
                        }
                    }
                    break;
                case '4':
                    if (arvore.eVazia()){
                        System.out.println("A árvore está vazia");
                    }else{
                        vetor = arvore.CamCentral();
                        String msg= " ";
                    }
                }
            }
        }
    }
}
```

```
        for (int i=0; i<arvore.getQuantNos();i++){
            msg+= vetor[i].getChave()+" ";
        }
        System.out.println("Exibir a árvore: "+ msg);
    }
    break;
case '5':
    System.out.println("fim do programa");
    break;
default:
    System.out.println("opção inválida, tente novamente");
}
} while (opcao!='5');
System.exit(0);
}
```

As árvores binárias de pesquisa (ABPs) representam uma ferramenta poderosa e versátil no arsenal da ciência da computação. Sua capacidade de organizar dados de forma hierárquica e eficiente permite que sistemas computacionais realizem operações de busca, inserção e remoção com notável desempenho.

A eficácia das ABPs reside em sua estrutura, que garante que, para cada nó, todos os valores na subárvore esquerda sejam menores e todos os valores na subárvore direita sejam maiores. Essa organização facilita a busca binária, um processo que divide o espaço de busca pela metade a cada passo, resultando em tempos de pesquisa logarítmicos no caso médio.

No entanto, é crucial reconhecer que o desempenho das ABPs pode variar dependendo da distribuição dos dados. Em cenários onde os dados são inseridos em ordem sequencial, a árvore pode degenerar em uma lista linear, comprometendo a eficiência da busca. Para mitigar esse problema, foram desenvolvidas variantes de ABPs, como árvores AVL e árvores rubro-negras, que buscam manter a árvore balanceada, garantindo um desempenho consistente.

Em suma, as árvores binárias de pesquisa desempenham um papel fundamental em sistemas computacionais que exigem operações eficientes de busca e manipulação de dados. Embora seja essencial considerar o potencial de desbalanceamento e suas implicações no desempenho, as ABPs, em suas diversas formas, continuam sendo uma ferramenta indispensável para a organização e o acesso rápido a informações em uma ampla gama de aplicações.

REFERÊNCIAS

- BROOKSHEAR, J. GLENN. **Ciência da computação: uma visão abrangente**. 7 ed. Porto Alegre: Bookman, 1999.
- BUCKNALL, Julian. **Algoritmos e Estruturas de Dados com Delphi**. SP: Editora Berkeley, 2002.
- CELES, W.; RANGEL, J. L. **10. Listas encadeadas**: documento de trabalho. São Paulo. Disponível em <<http://www.pessoal.fc.unesp.br/~erich/ed1/Listas.pdf>>. Acesso em 24 out. 2004.
- JUNIOR, LUIZ LIMA. **Árvores binárias**. Curitiba, 2004. Disponível em <<http://www.ppgia.pucpr.br/~laplima/aulas/materia/arvbin.htm>>. Acesso em 24 out. 2004.
- LISBÔA, JONIVAN COUTINHO. **Estruturas de dados**: documento de trabalho. Campos, 2002. Disponível em <<http://www.info.cefetcampos.br/~jonivan/ED-Tec/EstDados.doc>>. Acesso em 24 out. 2004.
- MORAES, Celso Roberto Moraes. **Estruturas de Dados e Algoritmos - Uma Abordagem Didática**. 2ª edição. SP: Editora Berkeley, 2003.
- PEREIRA, Silvio do Lago. **Estruturas de Dados Fundamentais: Conceitos e Aplicações**. SP: Editora Érica, 1996.
- PREISS, Bruno R.. **Estruturas de Dados e Algoritmos: Padrões de projetos orientados a objetos com Java**. RJ: Editora Campus, 2001.
- PUGA, Sandra; RISSETTI, Gerson. **Lógica de programação e estruturas de dados: com aplicações em Java**. São Paulo: Prentice Hall, 2004.
- RICARTE, IVAN L. M. **Estruturas de dados**. Campinas, 2003. Disponível em <<http://www.dca.fee.unicamp.br/~ricarte/ProgSist/node10.html>>. Acesso em 24 out. 2004.
- SHAHBAZKIA, HAMID REZA. **Filas!**. Portugal, 2001. Disponível em <<http://w3.ualg.pt/~hshah/ped/Aula%206/filas.html>>. Acesso em 23 out. 2004.
- SZWARCFTER, Jayme Luiz et al. **Estruturas de Dados e seus Algoritmos**. 2ª edição. RJ: Editora LTC- Livros Técnicos e Científicos, 1994.
- TENENBAUM, Aaron M. et al. **Estruturas de Dados usando C**. SP: Editora Makron Books, 1995.
- VASCONCELOS, JOSÉ; REIS, LUÍS PAULO. **Programação I – Introdução à algoritmo e estruturas de dados**: documento de trabalho. Fernando Pessoa, 2002. Disponível em <<http://piano.dsi.uminho.pt/ieee/repos/AEDados.pdf>>. Acesso em 24 out. 2004.
- VELOSO, Paulo et al. **Estruturas de dados**. 4ª edição. RJ: Editora Campus, 1986.
- VILLAS, MARCOS VIANNA et al. **Estruturas de dados: conceitos e técnicas de implementação**. Rio de Janeiro: Campus, 1993.
- WIRTH, Nilda. **Algoritmos e Estrutura de Dados**. Prentice do Brasil, 1999.
- ZIVIANI, Nívio. **Projeto de algoritmos com implementação em Pascal e C**. 2ª ed. SP: Editora Pioneira, 2004.