**Estrutura**

**De**

**Dados**

***Olavo Rafael Pereira Santos***

**1-Dados**

***1.1 Conceitos básicos de dados***

A natureza parece encontrar forma práticas de disposição de suas estruturas as quais, muitas vezes, são “copiadas” pelo homem. Um exemplo prático é uma árvore. Nela temos o tronco, o qual sustenta os galhos que, por sua vez, sustentam as folhas. O caminho percorrido pela seiva para atingir cada uma delas é relativamente pequeno, se pensarmos na distância que teria se os galhos fossem dispostos em uma fileira. Essa estrutura de ramos permite que cada folha seja atingida com o menor esforço, uma característica muito importante para a sobrevivência. Podemos também observar no exemplo anterior certa padronização. As folhas e os frutos de uma árvore são praticamente idênticos e, assim, cada estrutura tem sua uniformização. Essas características podem ser utilizadas em estruturas para armazenamento de informações, como, por exemplo, uma árvore genealógica.

Estamos dizendo isso, pois, nesta aula você verá a importância de organizar adequadamente as informações que serão manipuladas e que os computadores devem ser preparados para receberem as informações que serão tratadas pelos programas. Os programas de computadores que serão desenvolvidos devem estabelecer um contrato com o sistema operacional do computador. As informações que serão tratadas devem respeitar os tipos definidos nesse contrato, isto é, quando você definir que irá trabalhar com números em um determinado momento, apenas valores numéricos (existem vários tipos numéricos) deverão ser informados. Para tanto, você será apresentado aos tipos mais comuns de dados suportados pela linguagem de programação (será utilizado o JAVA) como caracteres, números, datas, valores booleanos e estruturas que podem conter mais de um tipo e como as informações são representadas em um programa e armazenadas na memória do computador através de variáveis.

Estrutura de dados é um modo de armazenar e organizar dados em um computador, de modo que possam ser usados eficientemente. A diferença entre dado e informação é que o dado é um valor sem sentido, ainda não processado, e informação, é mais detalhada, pois foi trabalhada com os elementos brutos. Por exemplo, o número 15 é o dado e a informação é 15 reais.

Quando os dados estão dispostos (organizados) de forma coerente, há uma estrutura de dados.

***1.1.1 Introdução à estrutura de dados***

Vamos iniciar a nossa a aula com uma suposição. Considere que você foi designado para organizar os registros dos empregados de uma empresa. Sua primeira missão é criar uma estrutura que contenha todos os nomes dos empregados da empresa. Para começar, você faz uma lista dos empregados com suas funções, conforme quadro abaixo:

Quadro 1 - Estrutura de Registros

| **Nome** | **Função** |
| --- | --- |
| João | Gerente |
| Pedro | Supervisor |
| Ana | Supervisor |
| Maria | Programador |
| Carla | Programador |
| Carlos | Programador |
| Marcos | Programador |

A estrutura acima nos dá apenas uma visão parcial da empresa. Não saberemos, por exemplo, os relacionamentos entre empregados, isto é, quem são seus superiores. Não temos uma visão de quais gerentes são responsáveis por quais empregados e assim por diante. Após pensar sobre o problema por um tempo, você decide que um diagrama é a estrutura melhor para representar os relacionamentos dos empregados na empresa.

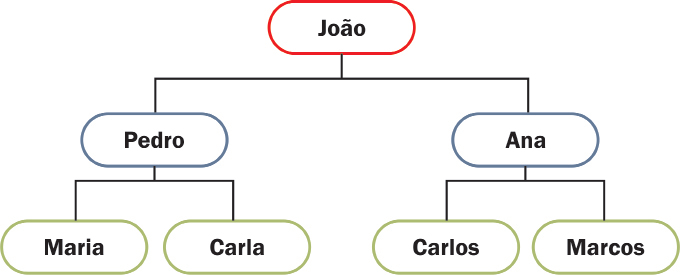


Figura 1 - Diagrama hierárquico da empresa

Descrição da imagem: diagrama hierárquico em que o nível 1, mais alto, é ocupado por João; o segundo nível é ocupado por Pedro e Ana; e o terceiro nível é ocupado por Maria, Carla, que derivam de Pedro, e Carlos e Marcos, que derivam de Ana.

Os diagramas acima são exemplos de estruturas de dados diferentes, já que na primeira representação temos os dados organizados em uma lista.

No caso da lista, os nomes dos empregados podem ser armazenados na ordem alfabética de modo que possamos encontrar o registro de um deles muito rapidamente. Entretanto, para exibir a relação entre os empregados, esta estrutura não é muito útil. A segunda estrutura (representando uma árvore) é mais adequada para esta finalidade.

Como você pode perceber, as estruturas com as quais representaremos os dados poderão promover uma organização mais adequada e, consequentemente, facilitar suas consultas. Precisamos estar atentos, pois existem várias maneiras de organizar os dados em estruturas e cada uma será mais indicada ao tipo de informação trabalhada. Por isso, para podermos conhecer quais as estruturas disponíveis, precisaremos conhecer primeiro como uma informação simples deve ser manipulada.

***1.1.2 Tipos de dados***

Os tipos de dados definem uma combinação de valores que uma variável pode receber e o conjunto de operações que esta pode executar. Os valores expressos podem ser de forma genérica, numéricos (“1”, “25”, “5,3”), caracteres (“a”, “abc”, “João”), datas (“10/10/2012”, “20 de maio de 2010”) etc.

Precisamos lembrar que as operações realizadas entre números do tipo inteiro são diferentes dos números reais (com casas decimais), os quais podem ser divididos. O mesmo vale para tipos como caracteres que não possuem operações matemáticas, mas podem, por exemplo, serem concatenados (“ca” + “sa” = “casa”). Como as variáveis dependem do sistema operacional e da linguagem utilizada, elas podem possui muitas variantes.

É importante salientar que definir o tipo de dado da informação que será manipulada é muito importante para o SO (sistema operacional) determinar o espaço que será reservado, pois é através dele que o SO solicitará uma porção da memória para armazenar a informação tratada.

Outra questão é que obter a informação da memória também depende do tipo de dado, pois são eles que possibilitam ao compilador realizar as conversões dos dados em memória para obter os valores manipulados nos programas. Lembre-se, como foi apresentado em Fundamento da Informática, que os dados em memória são representados apenas por “0” e “1” e é através do tipo de dado que esse conjunto de números binários são definidos como uma letra ou um número.

Imagine que eu lhe peça para buscar um recipiente para armazenar uma “coisa”. Sem saber o que é, você poderia pegar uma garrafa de 2 litros ou um pequeno saco de papel. Se eu pretendia lhe entregar um litro de água, a garrafa atenderia, mas o saco de papel não iria servir. Agora imagine que a quantidade poderá ser imensa e, desta forma, ficaria impossível idealizar o recipiente correto. Isso vale para o SO, quando um sistema vai manipular informações. Elas devem ser tipificadas para que não ocorra uma reserva de espaço em memória insuficiente ou extremamente grande.

Agora que já compreendemos a importância de definirmos o tipo de dados, vamos falar sobre as variáveis.

***Tipo de dado primitivo***

Vimos que os tipos de dados são uma combinação de valores que uma variável pode armazenar, o que pode variar conforme o sistema operacional e a linguagem de programação. Assim, uma variável nomeada “nome” utilizada para armazenar nomes de pessoas deverá estar associada a um tipo de dado compatível com os valores armazenados.

De acordo com a linguagem de programação utilizada, o tipo de um dado é verificado de maneira diferente, conforme a análise do compilador. O Java é uma linguagem compilada e possui seu conjunto de tipos de dado básico. Esses tipos são conhecidos como tipos primitivos, pois são suportados diretamente pelo compilador e são utilizados durante a codificação na definição de variáveis.

Veja o quadro abaixo onde temos os tipos de dados, sua descrição e seu tamanho.

Quadro 1 - Tipo de dado, descrição e tamanho

| **Tipo** | **Descrição** | **Tamanho** |
| --- | --- | --- |
| boolean | Pode assumir o valor true (verdadeiro) ou o valor false (falso) | 1 bit |
| char | Caractere em notação Unicode. Armazena dados alfanuméricos | 16 bits |
| Byte | Inteiro. Pode assumir valores entre -128 e 127. | 8 bits |
| Short | Inteiro. Valores possíveis: -32.768 a 32.767 | 16 bits |
| Int | Inteiro. Valores entre -2.147.483.648 e 2.147.483.647. | 32 bits |
| Long | Inteiro. Valores entre -9.223.372.036.854.770.000 e 9.223.372.036.854.770.000. | 64 bits |
| Float | Real (representa números em notação de ponto flutuante). Valores entre -1,4024E-37 e 3.40282347E + 38 | 32 bits |
| Double | Real (representa números em notação de ponto flutuante). Valores entre -4,94E-307 e 1.79769313486231570E + 308 | 64 bits |

O quadro acima representa alguns dos principais tipos de dados utilizados para manipulação de informações em programas desenvolvidos na linguagem JAVA. A coluna Tamanho especifica o espaço de memória utilizado por cada tipo. Sendo assim, é muito importante especificar adequadamente o tipo empregado para não reservarmos espaços de memória que não serão utilizados e, também, para não subdimensionarmos o espaço a ser utilizado.

***Tipos abstratos de dado***

Como vimos anteriormente, um tipo de dado é associado à forma de interpretação do conteúdo da memória do computador. Quando vamos trabalhar com valores inteiros temos um tipo de dados “int”.

Quando analisamos as informações sobre a perspectiva da forma com que as manipulamos como: somar, consultar etc. tem-se um conceito de Tipo de Dado chamado Tipo Abstrato de Dado (TAD). As estruturas de dados são uma maneira particular de se representar um TAD.

Exemplos de TAD:

* Listas;
* Pilhas e filas;
* Árvores.

Um TAD é uma especificação de uma coleção de dados e um grupo de operações que podem ser executadas sobre esses dados. Assim, um TAD descreve quais dados podem ser armazenados em uma única estrutura e o que é possível fazer com esses dados através das suas operações.

Os dados em um TAD devem ser protegidos de intervenções externas e apenas as operações especificadas no TAD podem manipular seus dados. Assim, o usuário de um TAD só tem acesso às operações disponibilizadas, as quais podem, por exemplo, consultar ou alterar os dados.

Vamos ver um exemplo:

Considere que você deseja representar um elevador com as seguintes características: capacidade de pessoas, total de pessoas no elevador e andar em que se encontra. Sem os TADs nós teríamos três variáveis distintas, mas com ele podemos ter a seguinte estrutura:

Quadro 2 - Representação de um Elevador

| **Elevador** |
| --- |
| int capacidade int totalPessoas int andar |
| Subir() Descer() EntrarPessoa() SairPessoa() |

Somente as funções definidas poderiam alterar os valores dos atributos e, assim, se desejássemos alterar os valores de totalPessoas, usaríamos as funções EntrarPessoa() para adicionar e SairPessoa() para retirar alguém.

As estruturas de dados são essenciais para a otimização da manipulação das informações tanto para o armazenamento quanto para a pesquisa/recuperação. Utilizar uma correta estrutura facilitará o desenvolvimento dos sistemas e os tornará mais eficientes. Existem várias estruturas e cada uma é mais adequada para um determinado conjunto de informações.

Manipular informações depende também das especificações correta dos tipos de dados, as quais serão representadas através das variáveis. O tipo de dado informa ao compilador que tipo de informação está sendo tratada, isto é, se os valores binários armazenados representam números, caracteres ou outros valores e permite que uma porção correta da memória seja reservada para o armazenamento da informação.

***Tipos de conjuntos de dados***

**Homogêneos**: Conjunto de dados formados pelo mesmo tipo de dado primitivo (vetores e matrizes), Canal TI (2015).

**Heterogêneos**: Tipos de dados primitivos diferentes (registros), Canal TI (2015).

***1.2 Vetores e matrizes***

Tente se imaginar em uma sala repleta de pastas, algumas contendo faturas, outras documentos de clientes e outras fichas de funcionários, espalhadas por todos os lados. Seu chefe entra e solicita que você informe o endereço de um cliente específico.

Parece uma tarefa muito difícil, principalmente se essa quantidade de pastas for maior que 1000 (mil). Verificar cada uma das pastas levaria muito tempo. Uma maneira de aperfeiçoar seu trabalho seria manter estas pastas em um arquivo, onde cada gaveta contivesse um grupo específico de pastas, uma gaveta para faturas, outra para clientes e outra para funcionários.

Sua tarefa agora ficaria um pouco mais fácil, já que teria que procurar o “cliente” na gaveta específica, o que diminuiria a quantidade de pastas para se consultar (ficaria muito mais fácil se estivessem em uma ordem específica, mas isso será tratado depois).

Existem estruturas que são semelhantes a uma pasta (uma variável), outras são semelhantes a uma gaveta do armário, onde se podem armazenar pastas contendo informações semelhantes (serão chamadas de vetores). Outras estruturas se assemelham ao próprio armário, contento várias gavetas (serão chamadas de matrizes). Uma matriz poderia representar até mesmo uma sala repleta de armários.

É importante que você faça essa associação, pois há situações em que nos vamos deparar com uma grande quantidade de variáveis do mesmo tipo de dado. Esperamos que, ao final desta aula, você seja capaz de manipular uma grande quantidade de variáveis do mesmo tipo de dado. Quando se tem que manusear muitos valores (considere, por exemplo, 20 (vinte) números inteiros informados pelo usuário), realizar o somatório ou identificar o maior seria um trabalho complicado, pois você teria que manipular 20 (vinte) variáveis, cada uma com um nome diferente. Teria que fazer vinte verificações para poder encontrar o maior número e o código ficaria muito extenso e complexo. Através das estruturas como vetores (arrays) e matrizes, essa tarefa se tornará bem mais simples, pois cada valor é associado a uma posição (parte) da estrutura e, com isso, tem-se apenas uma “variável”. Dessa forma, para acessar um valor, basta identificar a posição onde ele se encontra. Percorrer estas estruturas é mais fácil, o que torna as tarefas de busca dos valores bem mais simples.

***1.2.1 Vetores***

Um vetor ou array é uma estrutura de dado linear (uma única dimensão), composto por um determinado número (finito) de elementos (uma coleção de variáveis, as quais deverão ser do mesmo tipo de dado).

Em outras palavras, um vetor é um conjunto de elementos de um mesmo tipo de dado em que cada elemento desse conjunto é acessado através de um índice. O quadro abaixo representa um vetor de nomes. Nele têm-se os valores Maria, Carlos e Ana, sendo que o índice de Maria é 0, de Carlos é 1 e o de Ana é 2.

Quadro 1 - Vetor de Nomes

| **0** | **1** | **2** |
| --- | --- | --- |
| Maria | Carlos | Ana |

Para que possamos acessar os valores armazenados nesse vetor, temos que chamá-los pelo índice: 0, 1 ou 2.

Acessar os elementos de um vetor é muito rápido, sendo considerado o tempo constante, pois o acesso aos elementos é feito pelo seu índice. Entretanto, a operação de remoção de um elemento poderá ser complexa se for necessário que não existam espaços "vagos" no meio do vetor, pois nesse caso é necessário mover uma posição todos os elementos depois do elemento removido.

Por exemplo, para excluir o elemento Carlos:

Quadro 2 - Excluindo Elemento no Vetor

| **0** | **1** | **2** |
| --- | --- | --- |
| Maria |  | Ana |

Ficará a posição 1 vazia e, assim, teremos que deslocar os elementos a sua direita uma posição:

Quadro 3 - Vetor de Nomes

| **0** | **1** | **2** |
| --- | --- | --- |
| Maria | Ana |  |

Vamos aprender, agora, como declarar e manipular vetores

***Declarando vetores***

Para podermos manipular vetores, precisamos que inicialmente seja informado o tipo de dado dos elementos que serão armazenados. A declaração de um vetor pode-se diferenciar em cada linguagem de programação. Aqui, utilizaremos a linguagem JAVA para representar a declaração.

string[] nomes; // Declaração simples de um vetor chamado “nomes” que armazenará elementos do tipo “string”.

Depois de criado o vetor, devemos criar uma instância dele:

nomes = new string[3]; // Criada uma instância do vetor nomes informando que ele possuirá 3 posições.

Esta declaração poderá ser realizada em uma única linha:

string[] nomes = new string[3];

***Manipulando vetores***

A manipulação dos vetores se dá através das operações de inserção, consulta e remoção, muitas vezes sendo necessário percorrer os elementos do vetor para encontrar o elemento ou a posição desejada.

Vamos ver como isso funciona?

***Inserindo valores***

Para armazenar um valor em um vetor, é necessário fornecer um índice que indique a posição que esse elemento irá ocupar, por exemplo:

nomes[0] = "Maria";  
nomes[1] = "Carlos";  
nomes[2] = "Ana";

***Consultando valores***

Para consultarmos um valor do vetor na linguagem JAVA, basta informarmos o nome do vetor e o índice (sua posição no vetor), por exemplo:

string nome = nomes[2];

Este trecho de código irá atribuir a variável “nome” o valor “Ana” (se considerarmos as inserções realizadas anteriormente).

***Excluindo valores***

Para excluirmos valores de um vetor, basta atribuirmos um valor nulo (null) para a posição do vetor que contenha o elemento que desejamos remover ou uma string vazia (abrir e fechar aspas duplas).

nomes[2] = “”;

Este comando irá atribuir um espaço em branco à posição 2 do vetor. Assim, teremos o vetor com os seguintes elementos:

Quadro 4 - Excluindo Valores no Vetor

| **0** | **1** | **2** |
| --- | --- | --- |
| Maria | Ana |  |

Cuidados ao manipular vetores

* A posição do primeiro elemento de um vetor é indicada pelo índice (zero);
* O último elemento de um vetor de tamanho 10 (dez) é o de índice 9 (nove);
* Acessar uma posição inválida de um vetor causará um erro na execução de seu programa.

Percorrendo um vetor e listando seus valores

Para percorrer (acessar) todos os elementos de vetor, utilizaremos o comando “for”, conforme exemplo abaixo:

for (int i = 0; i < nomes.length; i++)  
 System.out.print(nomes[i]);

O comando nomes.length retorna o valor do tamanho do vetor, isto é, quantas posições ele possui. Em nosso exemplo esse valor é igual a três. Neste trecho de código, todos os elementos do vetor serão escritos na console do sistema.

Para realizarmos uma busca de um valor dentro do vetor, sem saber se o mesmo existe ou seu índice, por exemplo, se desejamos saber em qual índice do vetor está o nome Carlos, e se ele existe:

for (int i = 0; i < nomes.length; i++ )  
 if(nomes[i].toString() == "Carlos")  
 System.out.print("O nome Carlos está na posição: " + i);

Este código percorre o vetor e a cada volta ele compara se o valor é Carlos e, se for, ele escreve no console uma mensagem informando o índice em que se encontra a cor verde. Caso não exista, nada será escrito no console.

***1.2.2 Matrizes***

Matrizes são arranjos que podem possuir várias dimensões. Um quadrado desenhado em uma folha possui duas dimensões largura ou base e altura. Quando desenhamos um cubo, acrescentamos outra dimensão, a profundidade.

Um vetor, por ser representado em uma “linha”, é uma matriz de uma dimensão. Matrizes de duas dimensões, ou mais, possuem propriedades semelhantes a um vetor. A diferença é que devemos utilizar o número de índices para acessar um elemento, isto é, em uma matriz de duas dimensões (bidimensional – podemos associar ao quadrado) temos um índice para a base e outro para a altura.

Por convenção, o primeiro índice de uma matriz bidimensional (primeira dimensão), é nomeado como “linha”, enquanto o segundo índice (segunda dimensão) corresponde à “coluna” onde serão armazenados os elementos. Como vimos anteriormente, podemos comparar uma matriz bidimensional a um arquivo com gavetas, uma das dimensões representaria o número de cada gaveta, a outra dimensão representaria o número das pastas contidas em cada gaveta.

Uma matriz bidimensional de cinco linhas e cinco colunas:

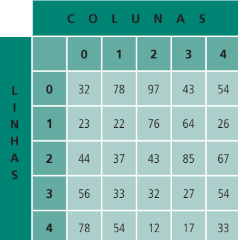


Figura 1 - Matriz Bidimensional

Descrição da imagem: Matriz Bidimensional mostrando colunas de números horizontais e linhas de números verticais. Nas horizontais temos na primeira coluna os números zero, um, dois, três, quatro. Na segunda, zero, trinta e dois, setenta e oito, noventa e sete, quarenta e três e cinquenta e quatro. Na terceira, um, vinte e três, vinte e dois, setenta e seis, sessenta e quatro e vinte e seis. Na quarta, dois, quarenta e quatro, trinta e sete, quarenta e três, oitenta e cinco, sessenta e sete. Na quinta, três, cinquenta e seis, trinta e três, trinta e dois, vinte e sete e cinquenta e quatro. Na sexta, quatro, setenta e oito, cinquenta e quatro, doze, dezessete e trinta e três.

Para podermos acessar o valor 22, localizamos o índice que será formado pela junção da linha “i” com a coluna “j”, i = 1 e j = 1, deste modo seu índice é [1,1]. O valor 54, por exemplo, será [0, 4].

Assim como os vetores, as matrizes têm o índice de cada linha, coluna ou qualquer outra dimensão iniciados com o valor 0 (zero). Por exemplo, o elemento posicionado na primeira linha (i=0) e na primeira coluna (j=0) da matriz terá índice [0,0].

Se acrescentarmos uma dimensão à matriz acima, teríamos uma tridimensional (cubo), a qual poderia representar uma sala com vários gaveteiros. Uma dimensão representaria os gaveteiros, outra representaria suas gavetas e, por último, a representação de pastas nas gavetas.

***Declaração de uma Matriz Bidimensional***

Veja como declarar uma matriz de duas dimensões:

int[] [] matriz1 = new int[2] [2];

Na sintaxe acima, declaramos uma matriz, de nome matriz1, de duas dimensões com duas linhas e duas colunas, ou seja, temos um arranjo com 4 posições:

Inicialização das posições da matriz:

matriz1[0][0] = 1;  
 matriz1[0][1] = 2;  
 matriz1[1][0] = 3;  
 matriz1[1][1] = 4;

Representação gráfica da matriz criada:

Quadro 1 - Representação gráfica da matriz criada

|  | **Coluna 0** | **Coluna 1** |
| --- | --- | --- |
| Linha 0 | 1 | 2 |
| Linha 1 | 3 | 4 |

***Percorrendo uma Matriz Bidimensional***

Vamos apresentar agora como percorrer cada posição da matriz:

for(int i = 0; i < 2; i++) {  
 for(int j = 0; j < 2; j++) {  
 System.out.print(matriz1[i][j]);  
 }  
}

O código acima mostra que precisamos de dois “for”, um dentro de outro. O primeiro percorre as “linhas” enquanto o interno percorre as “colunas”.

Vimos aqui que podemos organizar várias informações em uma única estrutura o que facilita a manipulação e diminui a complexidade do código. Um vetor é estruturalmente uma matriz de uma dimensão, isto é, possui uma linha e várias colunas e, para criarmos um vetor, precisamos informar o tipo de dados dos elementos que ele irá armazenar. O comando “for” é o mais indicado para se manipularem os vetores e as matrizes, pois sua estrutura facilita percorrer os elementos nestas estruturas.

***1.3 Estruturas lineares estáticas***

Para começarmos, vamos pensar naquilo com que lidamos todos os dias, de várias formas, sejam mais simples ou complexas: as listas.

As listas estão presentes em muitas atividades do dia a dia como, por exemplo, uma lista de compras de supermercado. Acrescentam-se nessa estrutura vários elementos, normalmente dispostos de cima para baixo, conforme figura abaixo:

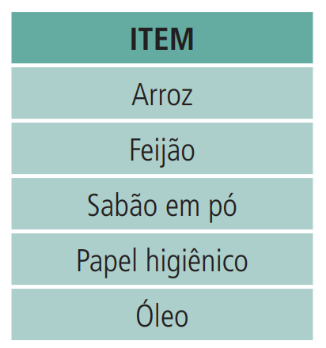


Figura 1 - Lista de Compras

Usualmente, você não se preocupa com a ordem dos elementos, já que vai inserindo conforme vai se lembrando das necessidades.

Essa representação de uma lista, com elementos em uma coluna, é normalmente utilizada em anotações. Porém, se imaginar essa lista “deitada” para a esquerda, ela ficará semelhante à estrutura de um vetor.

Quadro 1 - Lista de Compras

| Óleo | Arroz | Feijão | Batata |
| --- | --- | --- | --- |

A estrutura de vetor, para representar uma lista, é conhecida como lista simples, já que são mais fáceis de criar. O problema é realizar a exclusão de um elemento, pois demandará muito esforço.

Outra solução que você verá são as listas encadeadas, onde a ordem de pesquisa dos elementos é baseada em índices. Neste caso, a remoção de um elemento necessitará apenas de alterar o índice de ligação entre o elemento anterior e o posterior, diferente de uma lista simples que necessita realizar a movimentação de todos os elementos uma posição à esquerda.

***Introdução***

Como pudemos observar anteriormente, é simples criar as estruturas de dados que são similares ao modo em que a memória dos computadores é arranjada.

Podemos representar a memória de um computador como uma lista:



Figura 2 - Representação de memória RAM

Descrição da imagem: Representação de memória RAM. Mostrando alguns endereços e valores. Na primeira coluna estão os endereços, na ordem: um, dois, três, quatro e cinco. Na coluna de valores há os números um, vinte e três, quarenta e três, vinte e dois e cinco.

Os valores são armazenados em uma “lista” associados a um endereço que representa a posição onde se encontram.

A relação de empregados da companhia ACME é uma estrutura de dados linear. Como a memória do computador é também linear, é fácil ver como podemos representar esta estrutura como uma lista na memória.

Quadro 2 - Relação de Empregados

| Maria | João | Ana | Pedro | José |
| --- | --- | --- | --- | --- |

Podemos observar que a visão dos empregados da empresa ACME não é exatamente a mesma da lista original (representada na aula 01). Quando nós fazemos uma lista dos nomes, nós tendemos a organizar esta lista em uma coluna melhor que em uma fileira. Neste caso, a representação conceitual de uma lista é a coluna dos nomes. Entretanto, a representação física da lista na memória de computador é uma fileira.

Formalmente, uma lista é uma coleção de elementos, cada elemento da lista é comumente chamado de nó. Um nó da lista pode conter mais de um tipo de informação. Em uma lista simples o índice do elemento é utilizado para sua localização. Uma lista pode estar ordenada ou não.

Uma Lista L de tamanho n pode ser vista como sendo uma sequência de elementos da seguinte forma L = a1, a2, ..., an, onde:

* ai+1 é sucessor de ai (i < n);
* ai-1 precede ai (i > 1);
* uma lista de tamanho 0 é chamada nula ou vazia.

Exemplos:

* Dias da semana: segunda, terça, ..., domingo;
* Cartas do baralho (um naipe): dois, três, ..., reis, ás.

***Operações***

Existe um conjunto de operações básicas que são aplicadas a uma lista para manipular seus elementos. São elas:

* inicializar a lista;
* encontrar a posição de um determinado elemento;
* verificar se a lista é vazia (nula);
* inserir um elemento na lista;
* remover um elemento da lista; e
* percorrer a lista.

***Formas de representação***

As listas podem ser divididas de acordo com a forma com que são dispostas na memória do computador:

* **sequencial (simples)**: os elementos são armazenados sequencialmente na memória do computador; e
* **ligada (encadeada)**: os elementos não obedecem a uma sequência na memória. Existe um componente responsável pela ligação entre os elementos da lista.

***1.3.1 Lista sequencial (simples)***

Uma Lista Linear Simples é uma coleção de elementos os quais são dispostos linearmente na memória.

Uma lista L:[a1, a2, ..., an], com n > 0, obedece às seguintes propriedades:

* a1 é o primeiro elemento;
* an é o último elemento; e
* ak, 1<k>n, é precedido pelo elemento ak-1 e seguido pelo elemento ak+1.

Uma representação sequencial permite um rápido acesso ao i-ésimo termo da lista uma vez que podemos utilizar diretamente a capacidade de indexação dos vetores.

***Implementação de lista simples***

De acordo com a definição acima, pode-se dizer que uma lista linear é um vetor de uma dimensão. Assim, a implementação de uma lista é semelhante à implementação de um vetor simples.

1. public class Lista {  
2. private int[] valores;  
3. private int tamanho;  
4. }

* Linha 1: temos a declaração da classe Lista a qual terá as seguintes propriedades:
* Linha 2: um vetor de inteiros chamado valores;
* Linha 3: o tamanho máximo da lista.

***Inicialização da lista***

Método para inicializar a lista e definir o seu tamanho.

1. public Lista(int tam) {  
2. if (tam > 0) {  
3. tamanho = tam;  
4. valores = new int[tamanho];  
5. }   
6. }

* Linha 4: cria o vetor de valores definindo seu tamanho.

***Inserção de elementos***

Na inserção deve-se:

* verificar se a posição é válida (está dentro do tamanho da lista);
* armazenar a informação X na posição desejada.

1. public int Insere(int valor, int pos) {  
2. if (pos >= 0 && pos < valores.length) {  
3. valores[pos] = valor;  
4. return valor;  
5. }  
6. else return -1;  
7. }

* Linha 2: verifica se a posição “pos” para inserir é válida (maior/igual a zero e menor que o tamanho da lista).
* Linha 3: insere o valor “valor” na posição “pos”.
* Linha 6: retorna o valor -1 se a posição for inválida (menor que zero ou maior/igual ao tamanho.

***Remoção de elementos***

Na remoção deve-se:

* Verificar se a posição é válida (está dentro do tamanho da lista);
* Atribuir o valor nulo para a posição desejada.

1. public int Remove(int pos) {  
2. int valor;  
3. if (pos > 0 && pos < valores.length) {  
4. valor = valores[pos];  
5. valores[pos] = 0;  
6. return valor;  
7. }  
8. else return -1;  
9. }

* Linha 3: verifica se a posição “pos” para inserir é valida (maior/igual a zero e menor que o tamanho da lista).
* Linha 5: atribui o valor nulo a posição “pos”.
* Linha 8: posição inválida.

***Iniciando a lista e resgatando valores***

Para iniciar a lista, basta instanciar um objeto, l1 e, para resgatar um valor da lista, basta informar o índice (sua posição na lista), como por exemplo:

public static void main(String[] args) {  
Lista l1 = new Lista(5);  
l1.Insere(1, 0);  
l1.Insere(2, 1);  
System.out.print(l1.valores[1]);  
}

O trecho de código acima iniciará uma lista “l1”, atribuirá os valores 1 e 2 às posições 0 e 1 respectivamente e escreverá na console o valor 2.

***Consultando um determinado valor***

Como realizarmos uma busca de um valor dentro da lista, sem saber se o mesmo existe? Por exemplo, desejamos saber em qual índice da lista está a cor verde e se ela existe:

for (int i = 0; i < valores.length; i++ )  
 if(valores[i] == ”verde”)  
 System.out.print("O valor está no ín-  
dice: " + i);

Este código percorre o vetor com o comando “for” e a cada laço ele compara se o valor é verde e, se for, ele escreve na console uma mensagem informando o índice em que se encontra. Se o valor não for encontrado, nada será escrito.

***1.3.2 Lista ligada (encadeada)***

As listas ligadas (ou encadeadas) são utilizadas para evitar que operações de inserção e remoção dos elementos tenham custo linear, isto é, para evitar a necessidade de deslocamentos de partes inteiras da lista.

As listas ligadas consistem em uma série de elementos que não estão necessariamente armazenados em posições contíguas da memória, os quais são denominados nós.

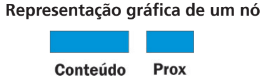


Figura 1 - Representação gráfica de um nó

Cada elemento da lista é composto por campo adicional (ponteiro) que contém o endereço do seu sucessor. O ponteiro da última estrutura aponta para NULO indicando o fim da lista ligada.

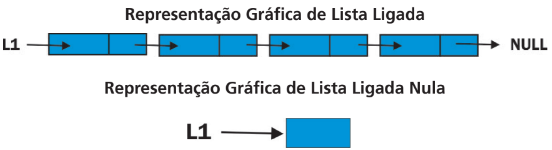


Figura 2 - Representações Gráficas de Lista Ligada e Nula

Para efetuarmos operações em listas ligadas, é necessário saber onde o primeiro elemento da lista se encontra, pois é a partir dele que podemos encontrar os outros elementos. Como os elementos não estão dispostos de maneira sequencial, utilizamos o ponteiro de cada elemento para saber onde está o próximo.

Considere a seguinte situação em que temos uma lista de pessoas onde cada pessoa sabe o endereço da próxima pessoa e sabemos, inicialmente, apenas onde mora a primeira pessoa da lista. Como fazer para encontrar uma determinada pessoa?

Se conhecermos o endereço da primeira pessoa, podemos ir até ela e perguntar qual o endereço da próxima pessoa já que ela tem essa informação. De tal modo, podemos ir até a próxima pessoa e perguntar o endereço da seguinte, assim sucessivamente até encontrar a pessoa desejada ou chegar ao final da lista se ela não existir.

Uma lista ligada estática é definida através de um vetor de duas dimensões com o tamanho máximo da lista representada. Uma das dimensões armazena a informação desejada. A outra dimensão armazena o campo ponteiro, um inteiro que contém o índice (endereço) do próximo elemento no vetor.

Veja a representação gráfica:

Quadro 1 - Lista Ligada Estática

| **Endereço** | **Informação** | **Ponteiro** |
| --- | --- | --- |
| 0 | João | 2 |
| 1 | Maria | 4 |
| 2 | Pedro | 1 |
| 3 |  |  |
| 4 | Carlos | -1 |

Se considerarmos que o primeiro elemento desta lista é o “João”, podemos percorrê-la da seguinte maneira:

João aponta para Pedro que aponta para Maria que aponta para Carlos que aponta para “nulo” (fim da lista).

Essa definição permite que espaços livres sejam criados na estrutura. Assim sendo, como gerenciar estes espaços livres?

Solução: criar uma lista de espaços livres.

Realizaremos uma implementação utilizando uma matriz de duas dimensões. Na coluna 0 (zero) são armazenados os valores e na coluna 1 (um) são armazenados os ponteiros.

**Observação:** nesta implementação a inserção de um elemento é realizada logo após o último elemento da lista, não se considerando os espaços livres que podem surgir após a realização de exclusões.

Utilizamos em nosso cotidiano uma infinidade de listas: compras, telefônica, clientes etc. Vimos nesta aula que as estruturas de vetores e matrizes podem ser utilizadas para representar uma estrutura linear denominada lista. Estas listas podem ser sequenciais ou ligadas. As listas sequenciais armazenam seus elementos respeitando a sequencialidade da memória do computador, enquanto as ligadas podem ter seus elementos armazenados em posições não contíguas da memória.

***1.4 Estruturas lineares dinâmicas***

Agora que você já domina a lista simples e a lista ligada estática, podemos avançar no nosso curso, trabalhando com as listas ligadas dinâmicas.

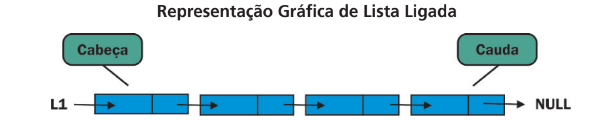
Para começarmos, lembre-se das listas sequenciais estáticas, vistas anteriormente. Elas requerem a definição inicial do tamanho (número de elementos que irá armazenar), o que engessa a estrutura. Quando a lista está cheia, não é possível inserir mais elementos e uma quantidade pequena de elementos fará com que o espaço de memória reservado para o restante da lista fique ocioso. Ainda, se desejar excluir elementos, a operação poderá se tornar muito complexa e custosa, pois exige uma grande quantidade de movimentações.

Neste tópico, vamos utilizar uma forma mais flexível de representar uma estrutura de lista, as listas ligadas dinâmicas. Essa estrutura não exige a definição de tamanho inicial, permitindo que inserções possam ser realizadas limitadas apenas pela disponibilidade de memória. Outra facilidade é a remoção de elementos e até mesmo a inserção de elementos no meio da lista, que não exige a movimentação dos outros elementos.

***1.4.1 Lista ligada dinâmica***

Uma lista ligada (ou encadeada) dinamicamente é uma estrutura de dados linear e dinâmica. Ela é composta por um conjunto de nós, cada nó possuindo dois elementos: o primeiro armazena informações e o segundo armazena o ponteiro, o qual guarda informação do endereço (referências a endereços de memória) para o próximo nó na lista.

Em uma implementação de lista ligada, cada nó (item) da lista é ligado com o seguinte através do ponteiro (uma variável com o valor do endereçamento para o próximo nó). Isso significa dizer que cada nó da lista contém o registro de dados (informações) e uma variável que aponta para o próximo nó. Este tipo de implementação permite utilizar posições não contíguas de memória, sendo possível inserir e retirar elementos sem haver a necessidade de deslocar os nós seguintes da lista.



A cabeça em uma lista ligada dinamicamente representa o primeiro elemento da lista e deve ser guardada em uma variável (primeiro) e a cauda representa o último elemento a qual também deve ser guardada em uma variável (ultimo). Assim, conhecemos, inicialmente, dois elementos da lista: o primeiro e o último. Se desejarmos manipular a lista, devemos inicialmente nos referenciar a estas variáveis e, a partir delas, percorrer a lista para identificar o restante.

O limite para a alocação dinâmica é diretamente proporcional à quantidade de memória física que está disponível na máquina, valendo, também, o tamanho do espaço físico disponível dentro do sistema de memória virtual

Se a memória disponível for insuficiente, a expressão lança um OutOfMemoryError.

Vamos ver as vantagens e desvantagens da utilização desse tipo de lista, segundo o portal Tol:

**Vantagens**

A inserção ou remoção de um elemento na lista não implica a mudança de lugar de outros elementos.

Não é necessário definir, no momento da criação da lista, o número máximo de elementos que esta poderá ter. em outras palavras, é possível alocar memória "dinamicamente", apenas para o número de nós necessários.

**Desvantagens**

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.

Para aceder ao elemento na posição n da lista, deve-se percorrer os n - 1 anteriores.

Conhecedores dos prós e contras, vamos aprender como implementar uma lista ligada dinâmica.

Exemplo prático de uma lista ligada dinâmica

Uma boa maneira de entender o funcionamento de uma lista ligada dinâmica é pensar em uma corrente de elos. Cada elo está conectado a um próximo elo na corrente, formando uma sequência de elementos.

Por exemplo, imagine que você tem uma corrente de elos e quer adicionar uma bola a ela. Você pega a bola e a conecta ao último elo da corrente. Em seguida, você pode adicionar mais bolas à corrente, conectando cada bola ao último elo.

Assim como na corrente de elos, em uma lista ligada dinâmica, cada elemento é formado por um nó que contém o valor do elemento e um ponteiro que aponta para o próximo nó na sequência. O último nó da lista tem um ponteiro que aponta para NULL, indicando que não há mais nós na lista.

Por exemplo, imagine que você queira criar uma lista ligada dinâmica para armazenar os números 1, 2, 3 e 4. Você cria o primeiro nó com o valor 1 e um ponteiro que aponta para NULL. Em seguida, você cria o segundo nó com o valor 2 e um ponteiro que aponta para o terceiro nó. O terceiro nó tem o valor 3 e um ponteiro que aponta para o quarto nó. O quarto nó tem o valor 4 e um ponteiro que aponta para NULL.

Assim, a lista ligada dinâmica é formada por uma sequência de nós, cada um contendo um valor e um ponteiro para o próximo nó na sequência. Você pode adicionar ou remover elementos da lista ligada dinâmica modificando os ponteiros entre os nós, sem precisar realocar toda a lista como ocorre em um vetor estático.

***Implementação da lista ligada dinâmica***

A implementação a seguir utilizará os conceitos de Orientação a Objeto para criar a lista. Teremos duas classes: No e Lista.

Inicialmente, criamos uma Classe No e ela será a nossa base para a alocação dinâmica e representará cada elemento da lista. Contém 2 (dois) atributos fundamentais: o campo dado para receber a informação e prox para referenciar o próximo objeto da nossa lista.

public class No {  
 private String dado;  
 private No prox;

//Propriedades da classe  
 public No getProx() {  
 return this.prox;  
 }

public void setProx(No prox) {  
 this.prox = prox;  
 }

public String getDado() {

return this.dado;

}

//Construtor da classe No  
 public No(String dadonovo) {  
 dado = dadonovo;  
 prox = null;  
 }

public No(String dadonovo, No ligacao) {  
 dado = dadonovo;  
 prox = ligacao;  
 }  
}

Alguns conceitos importantes relacionados à atribuição de valores a objetos:

No n1 = new No(); //declara n1 como No, aponta para o nó criado n1. Dado = “aa”;

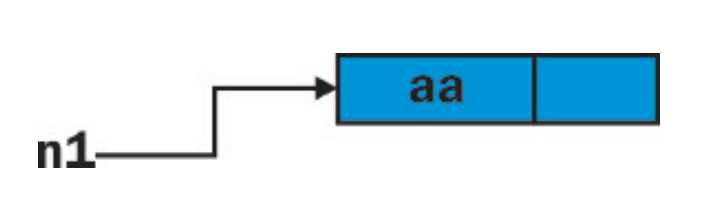


Figura 2 - Conceitos de atribuição de valores a objetos

No n2 = n1; //declara n2 como No e atribui o endereço de n1 a n2, tanto n1 quanto n2 apontam para o nó criado.

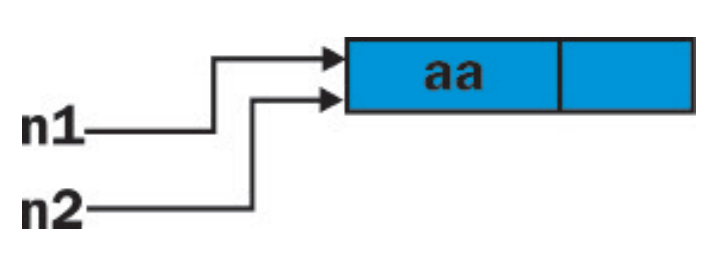


Figura 3 - Conceitos de atribuição de valores a objetos

No n3 = new No(); //declara n3 como No, aponta para o nó criado n3. Dado = “bb”;



Figura 4 - Conceitos de atribuição de valores a objetos

n1.Prox = n3; //atribui o endereço de n3 ao atributo Prox do nó n1. Assim o “prox” de n1 apontará para n3. O mesmo vale para n2, pois ele “aponta” para o mesmo endereço de n1.

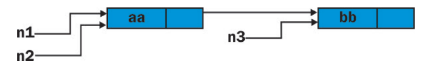


Figura 5 - Conceitos de atribuição de valores a objetos

A classe Lista possuirá três atributos principais: primeiro (indica o início da lista, declarado com o tipo No), último (indica o fim da lista, declarado com o tipo No) e nomeDaLista.

Perceba que a lista é “composta” apenas de dois nós. Na verdade, é o que basta para manipularmos a lista e não precisamos conhecer o restante, pois a partir do primeiro podemos percorrer toda a lista, seguindo o conceito de que cada nó conhece o seu sucessor. Assim, o primeiro sabe qual é o próximo, sucessivamente até chegarmos ao último.

public class ListaLigadaDinamica {

private No primeiro;

private No ultimo;

private String nomeDaLista;

//Construtor da classe Lista

public ListaLigadaDinamica(String nome) {

nomeDaLista = nome;

//Como a lista inicialmente é vazia, tanto o primeiro como o ultimo

// receberão o valor nulo.

primeiro = ultimo = null;

}

public ListaLigadaDinamica() {

nomeDaLista = "Lista Teste";

primeiro = ultimo = null;

}

}

Os métodos InsereNaFrente e InsereNoFundo acrescentam, respectivamente, um elemento no início da lista e no final da lista.

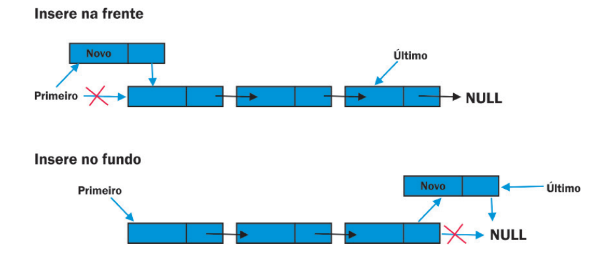


Figura 6 - Métodos InsereNaFrente e InsereNoFundo

Descrição da Imagem: Insere na frente: Primeiro vai para Novo e depois para o próximo retângulo até o Último e a saída (NULL). Insere no fundo: Primeiro vai direto para o retângulo e antes de sair pelo NULL vai para o retângulo Novo que é o Último e depois sai pelo NULL. Os métodos RemoveDaFrente e RemoveDoFundo, retiram, respectivamente, um elemento do início da lista e do final da lista.

public void InsereNaFrente(String item) {  
 //Verifica se a lista está vazia. Se estiver, o primeiro e o ultimo receberão o  
 // nó que está sendo criado.  
 if (Vazia())  
 primeiro = ultimo = new No(item);  
 //Quando a lista não está vazia o primeiro recebe o nó criado e seu “prox”  
 // recebe o”antigo” primeiro.  
 else  
 primeiro = new No(item, primeiro);  
 }  
 public void InsereNoFundo(String item) {  
 if (Vazia())  
 primeiro = ultimo = new No(item);  
 else

ultimo.setProx(new No(item));

ultimo = ultimo.getProx();

}

}

Os métodos RemoveDaFrente e RemoveDoFundo, retiram, respectivamente, um elemento do início da lista e do final da lista.

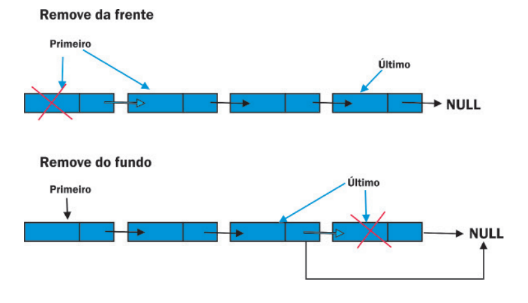


Figura 7 - Métodos RemoveDaFrente e RemoveDoFundo

Descrição da Imagem: Remove da frente: Não inicia pelo primeiro retângulo, e sim, pelo segundo, passa pelo terceiro e no quarto sai pelo NULL. Remove do fundo: Inicia pelo primeiro retângulo, passa pelo segundo e já sai pelo terceiro (NULL).

***Implementação da Lista Ligada Dinâmica***

A implementação a seguir utilizará os conceitos de Orientação a Objeto para criar a lista. Teremos duas classes: No e Lista.

Inicialmente, criamos uma Classe No e ela será a nossa base para a alocação dinâmica e representará cada elemento da lista. Contém 2 (dois) atributos fundamentais: o campo dado para receber a informação e prox para referenciar o próximo objeto da nossa lista.

public class No {  
 private String dado;  
 private No prox;

//Propriedades da classe  
 public No getProx() {  
 return this.prox;  
 }

public void setProx(No prox) {  
 this.prox = prox;  
 }

public String getDado() {  
 return this.dado;  
 }

//Construtor da classe No  
 public No(String dadonovo) {  
 dado = dadonovo;  
 prox = null;  
 }

public No(String dadonovo, No ligacao) {  
 dado = dadonovo;  
 prox = ligacao;  
 }  
 }

Alguns conceitos importantes relacionados à atribuição de valores a objetos:

No n1 = new No(); //declara n1 como No, aponta para o nó criado n1. Dado = “aa”;



Figura 8 - Atribuição de valores a objetos

No n2 = n1; //declara n2 como No e atribui o endereço de n1 a n2, tanto n1 quanto n2 apontam para o nó criado.

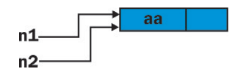


Figura 9 - Atribuição de valores a objetos

No n3 = new No(); //declara n3 como No, aponta para o nó criado n3.Dado = “bb”;



Figura 10 - Atribuição de valores a objetos

n1.Prox = n3; //atribui o endereço de n3 ao atributo Prox do nó n1. Assim o “prox” de n1 apontará para n3. O mesmo vale para n2, pois ele “aponta” para o mesmo endereço de n1.

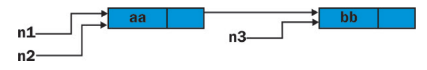


Figura 11 - Atribuição de valores a objetos

A classe Lista possuirá três atributos principais: primeiro (indica o início da lista, declarado com o tipo No), último (indica o fim da lista, declarado com o tipo No) e nomeDaLista.

Perceba que a lista é “composta” apenas de dois nós. Na verdade, é o que basta para manipularmos a lista e não precisamos conhecer o restante, pois a partir do primeiro podemos percorrer toda a lista, seguindo o conceito de que cada nó conhece o seu sucessor. Assim, o primeiro sabe qual é o próximo, sucessivamente até chegarmos ao último.

public class ListaLigadaDinamica {  
 private No primeiro;  
 private No ultimo;  
 private String nomeDaLista;

//Construtor da classe Lista  
 public ListaLigadaDinamica(String nome) {  
 nomeDaLista = nome;  
 //Como a lista inicialmente é vazia, tanto o primeiro como o ultimo  
 // receberão o valor nulo.  
 primeiro = ultimo = null;  
 }

public ListaLigadaDinamica() {  
 nomeDaLista = "Lista Teste";  
 primeiro = ultimo = null;  
 }  
 }

Os métodos InsereNaFrente e InsereNoFundo acrescentam, respectivamente um elemento no início da lista e no final da lista.

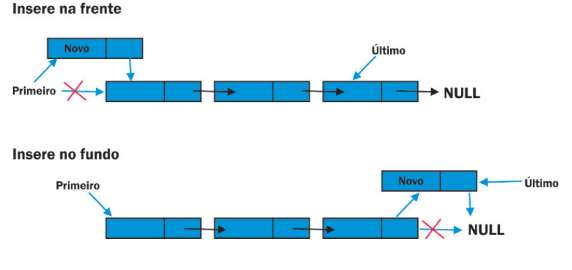


Figura 12 - O que o InsereNaFrente e InsereNoFundo acrescentam

Descrição da Imagem: Insere na frente: Passa pelo Novo retângulo e depois pelo segundo, terceiro e quarto retângulos até sair pelo NULL. Insere no fundo: Entra no primeiro retângulo, passa pelo segundo e por fim em um Novo, saindo pelo NULL.

public void InsereNaFrente(String item) {  
 //Verifica se a lista está vazia. Se estiver, o primeiro e o ultimo receberão o  
 // nó que está sendo criado.  
 if (Vazia())  
 primeiro = ultimo = new No(item);  
 //Quando a lista não está vazia o primeiro recebe o nó criado e seu “prox”  
 // recebe o”antigo” primeiro.  
 else  
 primeiro = new No(item, primeiro);  
 }

public void InsereNoFundo(String item) {  
 if (Vazia())  
 primeiro = ultimo = new No(item);  
 else  
 {  
 ultimo.setProx(new No(item));  
 ultimo = ultimo.getProx();  
 }  
 }

Os métodos RemoveDaFrente e RemoveDoFundo, retiram, respectivamente, um elemento do início da lista e do final da lista.

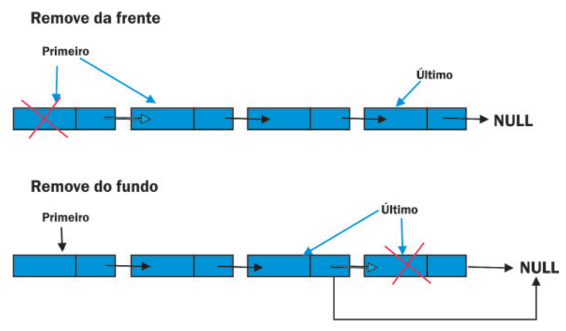


Figura 13 - O que RemoveDaFrente e RemoveDoFundo retiram

Descrição da imagem - Remove da frente: Passa pelo segundo, terceiro e o quarto sai (NULL).Remove do fundo: Passa pelo primeiro, segundo e terceiro sai (NULL).

 public String RemoveDaFrente() {  
        //Verifica se está vazia, s e sim, não faz nada e retorna nulo.  
 if (Vazia())  
 {  
 return null;  
 }  
 String item = primeiro.getDado();  
 //Verifica se o primeiro é igual ao ultimo, se sim, existe apenas um  
 // elemento na lista. Atribui o valor nulo para ambos.  
 if (primeiro == ultimo)  
 primeiro = ultimo = null;  
 //Caso contrário, atribui o endereço do segundo nó à variavel primeiro.  
 else  
 primeiro = primeiro.getProx();  
 return item;

}

public String RemoveDoFundo() {  
 if (Vazia())  
 {  
 return null;  
 }  
 String item = ultimo.getDado();  
 if (primeiro == ultimo)  
 primeiro = ultimo = null;  
 else  
 {  
 No atual = primeiro;  
 while (atual.getProx() != ultimo)  
 atual = atual.getProx();  
 ultimo = atual;  
 atual.setProx(null);  
 }  
 return item;  
 }

O Método Vazia retorna verdadeiro se a lista está vazia e falso caso contrário.

public boolean Vazia() {  
 //Se a variável primeiro é nula, então a lista não possui elementos (nós), ela está vazia  
 return primeiro == null;  
 }

O Método EscreveLista percorre toda a lista retornando todas as informações dos nós da lista. Para percorrer uma lista ligada, usamos o atributo prox de cada nó. Considere o seguinte exemplo:



Figura 14 - Método EscreveLista

A lista L1 possui os nós: n1 => primeiro e n4 = ultimo. Os nós são ligados da seguinte maneira:

primeiro == n1  
 n1.prox => n2  
 n2.prox => n3  
 n3.prox => n4  
 n4.prox => null  
 ultimo == n4

Como conhecemos o início da lista, a variável primeiro representa-o, basta seguí-la, através do “prox”, até o último. Para isso, normalmente utilizamos uma variável (atual) que vai sendo associada a cada nó da lista.

No atual = primeiro; //A variável atual aponta para o nó n1, o início da lista.

atual = atual.prox; //Como atual aponta para n1, atual.prox aponta para n2. Assim, atual passará a apontar para n2.

Realizando o comando acima, sucessivamente, vamos caminhando pela lista até o último.

public String EscreveLista() {  
 String temp = "";  
 if (Vazia()) {  
 temp += "Vazia " + nomeDaLista;  
 return temp;  
 }//if vazia  
 temp += "A " + nomeDaLista + " contem \n\n";  
 No atual = primeiro;  
 while (atual != null) {  
 temp += atual.getDado() + ", ";  
 atual = atual.getProx();  
 }  
 temp += "\n";  
 return temp;  
 }

O método exibePrim e exibeUlt, exibem, sucessivamente, o primeiro e o último elemento da lista.

public String exibePrim() {  
 if (primeiro == null) return "Primeiro: null";  
 else return "Primeiro: " + primeiro.getDado();  
 }  
 public String exibeUlt() {  
 if (ultimo == null) return "Ultimo: null";  
 else return "Ultimo: " + ultimo.getDado();  
 }

Teste de Classe

public static void main(String[] args) {  
 ListaLigadaDinamica l1 = new ListaLigadaDinamica();  
 l1.InsereNaFrente("no1");  
 l1.InsereNaFrente("no2");  
 l1.InsereNoFundo("no3");  
 System.out.print(l1.EscreveLista());  
 l1.RemoveDaFrente();  
 l1.RemoveDoFundo();  
 System.out.print(l1.EscreveLista());  
 }

Saída do comando System.out.print(l1.EscreveLista()): no2, no1, no3, Saída do comando System.out.print(l1.EscreveLista()): no1.

As listas ligadas facilitam as operações de inserção ou remoção dos elementos (nós), pois não são necessários os deslocamentos dos demais elementos.

Ao criarmos as listas ligadas dinâmicas, não é necessário definir o seu tamanho, isto é, o número máximo de elementos que ela poderá conter. Seu tamanho é limitado pela disposição de espaço na memória, uma vez que cada elemento é adicionado "dinamicamente".

***1.4.2 Lista duplamente ligada***

As listas duplamente ligadas (ou listas duplamente encadeadas) são uma extensão da lista simples ligada apresentada no item anterior.

Uma lista duplamente ligada é formada por um conjunto de nós composto normalmente por três elementos, uma variável que armazena a informação, podendo ser objetos, números, caracteres etc. e dois ponteiros que possibilitam a ligação entre os nós anterior e posterior desta lista.

Assim, enquanto em uma lista simples ligada, cada nó (elemento) conhece (aponta) apenas o próximo nó, nas listas duplamente ligadas, os nós conhecem dois outros nós, seu antecessor e seu sucessor, com exceção da cabeça e da cauda em que o primeiro aponta apenas para o seu sucessor e o segundo aponta apenas para o seu antecessor.

Este tipo de solução permite percorrer uma lista nas duas direções. Por exemplo, do início (CABEÇA) até o final da lista (CAUDA), utilizaremos o ponteiro PRÓXIMO. Para percorrer a lista do final até o início (em ordem inversa), devemos começar com o último nó (CAUDA) e, através do ponteiro ANTERIOR, percorrer a lista até encontrar o primeiro nó (CABEÇA).

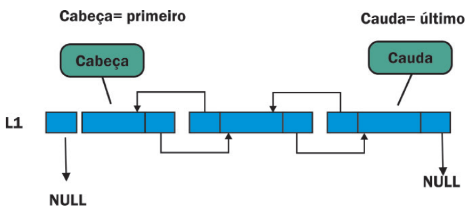


Figura 1 - Representação Gráfica de Lista Duplamente Ligada

Descrição da Imagem: Lista Duplamente Ligada. L1 entra pelo NULL, passa pela cabeça e pelo meio e sai pelo NULL (cauda). Cabeça igual a primeiro. Cauda igual a último. Setas apontam do primeiro até o último e do último até o primeiro, mostrando o trajeto de L1.

Esta lista é uma variação da lista duplamente ligada (encadeada), onde o ponteiro “prox” do último elemento aponta para o primeiro elemento da lista e o ponteiro “ant” do primeiro elemento aponta para o último elemento da lista formando um círculo.

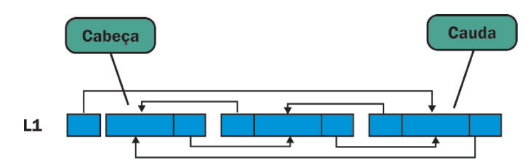


Figura 2 - Representação gráfica da Lista Circular

As listas estão presentes em praticamente todas as organizações e sempre nos utilizamos de suas funcionalidades. Vimos nesta aula que existem estruturas de dados que são capazes de representar todas essas listas, as quais não necessitam ter seu tamanho especificado inicialmente e que são chamadas de listas ligadas ou encadeadas. Estas listas armazenam seus elementos em posições que não necessitam ser contíguas na memória do computador.

As listas ligadas facilitam operações de remoção e inserção de elementos, uma vez que não necessitam de deslocamentos dos seus nós para realizar tais operações.

As listas ligadas possuem duas variações principais: as listas duplamente ligadas, as quais mantêm o conhecimento dos nós adjacentes (anterior e posterior) e as listas circulares que têm como característica principal a ligação do último nó com o primeiro, formando um círculo.

**2-Filas e classificações de algorismos**

***2.1 Introdução***

Vamos começar a estudar este tópico com um “exemplo real”:

Um dos maiores dramas dos clientes que necessitam de serviços bancários ou dos contribuintes à procura de seus direitos em muitas repartições são as filas enormes e demoradas. Sem discutir a eficiência dos serviços prestados, as filas são essenciais para garantir o direito do atendimento das pessoas que chegaram primeiro. Sem elas, seria um caos identificar quais pessoas devem ser atendidas primeiro. A saída de uma fila deve obedecer à ordem de chegada, isto é, será atendido primeiro aquele que chegou primeiro e as novas pessoas que chegam devem ficar no final da fila.

Outra estrutura que muitas vezes é utilizada em nosso cotidiano é a pilha. Pilhas de cartas, pratos e caixas possibilitam que sejam retirados os itens de cima, mas quando colocamos um novo item, esse é posto, também, na parte de cima. Uma pilha de pratos de um restaurante self-service tem seus pratos retirados, pelos clientes, da parte de cima e, sempre que são repostos, eles são colocados na parte de cima.

Você saberá criar, ao final deste tópico, estruturas de dados que representam as pilhas e filas. Muito usadas para controle da ordem de utilização do processamento dos aplicativos, os sistemas operacionais trabalham com os conceitos de filas e pilhas. Utilizará os conceitos apresentados de listas ligadas para implementá-las. Assim, elas serão representadas através da ligação de nós, obedecendo às posições de retirada e inserção dos elementos, as quais sempre estarão nas extremidades.

***2.2 Filas***

As filas são estruturas que preservam a ordem de chegada, isto é, o primeiro a chegar será o primeiro a sair, estratégia chamada FIFO (first in, first out). Um jogo de cartas com baralhos será modelado como uma fila, se retirarmos as cartas em cima e colocarmos os descartes de volta por baixo. Exemplos do cotidiano são as filas para atendimento às pessoas.

As intervenções de inserção são realizadas no final da fila e as intervenções de remoção são realizadas no início. A inserção de um elemento torna-o último da fila. As operações básicas em uma fila são:

* Enfileira(x): insere o item x no fim da fila.
* Desenfileira(): retorna e remove o elemento do início da fila.
* Inicializa(): cria uma fila vazia.
* Cheia(), Vazia(): testa a fila para saber se ela está cheia ou vazia.

As filas são mais complexas de implementar do que as pilhas, porque as interações ocorrem nas duas extremidades (início e fim). A implementação mais simples usa um vetor, inserindo novos elementos no final e retirando do início, depois de uma remoção, movendo todos os elementos uma posição para a esquerda. Utilizaremos os conceitos de listas ligadas, vistos anteriormente, para implementação.

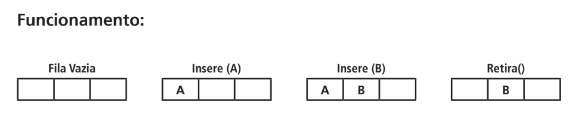


Figura 1 - Exemplo de fila

Descrição da imagem: Funcionamento: três quadradinhos, todos vazios (fila vazia). Insere (A): No primeiro quadradinho temos A. Insere (B): No primeiro quadradinho temos A e no segundo B. Retira (): Só temos B no segundo quadradinho.

Exemplo de fila na prática

Para entender o conceito de uma fila, pense em uma fila de pessoas em uma loja. Cada pessoa entra na fila no final e sai dela na frente, seguindo uma ordem específica.

Por exemplo, imagine que há uma loja que vende ingressos para um show. Há várias pessoas na fila, todas esperando sua vez de comprar um ingresso. A primeira pessoa que entrou na fila será a primeira a comprar um ingresso e sair da fila, seguida pela segunda pessoa, a terceira pessoa e assim por diante, até que todas as pessoas tenham comprado seus ingressos.

Essa é a essência da estrutura de dados de fila - ela é uma coleção de elementos organizados de forma que o primeiro elemento adicionado seja o primeiro a ser removido (conhecido como "First In, First Out" ou FIFO em inglês).

Por exemplo, se você adicionar os elementos 1, 2, 3 e 4 em uma fila, o elemento 1 será o primeiro a ser removido, seguido pelo 2, 3 e 4, na ordem em que foram adicionados.

Assim como na fila da loja, em uma fila de dados, você só pode adicionar elementos ao final da fila e remover elementos do início da fila. Além disso, você não pode acessar elementos que estejam no meio da fila sem antes retirar os elementos que estão à frente deles.

Implementação de uma fila

Normalmente, as filas são implementadas através da utilização de vetores, mas essa implementação é complexa e tem algumas desvantagens de acordo com o método adotado. Um vetor simples necessitaria de deslocamento de todos os elementos, uma posição à esquerda, para cada retirada da fila. Outra solução emprega uma lista circular no vetor e necessita de controlar os índices à medida que cada elemento é retirado ou inserido na fila.

Vamos utilizar uma solução mais prática e elegante, através de uma lista ligada. Como nestas listas utilizamos as variáveis nomeadas como primeiro e último, fica bem simples adaptá-las para uma fila. Apenas temos que considerar as operações insere na frente e remove do fundo, as outras devendo ser descartadas Inicialmente criamos uma Classe No, que representará cada elemento da fila.

Inicialmente criamos uma Classe No, que representará cada elemento da fila.

public class No {  
 ver implementação no item 4.1.1  
 }

A classe Fila possuirá os três atributos principais das listas: primeiro (indica o início da fila, declarado com o tipo No), último (indica o fim da fila, declarado com o tipo No) e nomeDaFila.

public class Fila {  
 private No primeiro;  
 private No último;  
 private String nomeDaFila;

//Construtor da classe Fila  
 public Fila(String nome) {  
 nomeDaFila = nome;  
 //Como a fila inicialmente é vazia, tanto o primeiro como o último  
 // receberão o valor nulo.  
 primeiro = ultimo = null;  
 }

public Fila() {  
 nomeDaFila = "Fila Teste";  
 primeiro = ultimo = null;  
 }  
 }

Não temos um método para inserir elementos na frente, apenas o método Enfileira que acrescentará os elementos no final da fila.

public void Enfileira(String item) {  
 if (Vazia())  
 primeiro = ultimo = new No(item);  
 else {  
 ultimo.setProx(new No(item));  
 ultimo = ultimo.getProx();  
 }  
 }

Aqui também não temos um método para retirar elementos do fim, apenas o método Desenfileira que removerá o primeiro elemento.

public String Desenfileira() {  
 //Verifica se está vazia, se sim, não faz nada e retorna nulo.  
 if (Vazia()){  
 return null;  
 }  
 String item = primeiro.getDado();  
 //Verifica se o primeiro é igual ao último, se sim, existe apenas um  
 //elemento na fila. Atribui o valor nulo para ambos.  
 if (primeiro == ultimo)  
 primeiro = ultimo = null;  
 //Caso contrário, atribui o endereço do segundo nó à variavel primeiro.  
 else  
 primeiro = primeiro.getProx();  
 return item;  
 }

O Método Vazia retorna verdadeiro se a fila está vazia e falso caso contrário.

public boolean Vazia() {  
 //Se a variável primeiro é nula, então a fila não possui elementos (nós), ela está vazia.  
 return primeiro == null;  
 }

Teste da Classe

public static void main(String[] args) {  
 Fila f1 = new Fila();  
 f1.Enfileira("el01");  
 f1.Enfileira("el02");  
 System.out.print(f1.Desenfileira());  
 }

Saída: el02

***2.2.1 Exercício sobre filas***

Suponha que você está trabalhando em um sistema de atendimento ao cliente, onde os usuários podem enviar solicitações para um suporte técnico. Cada solicitação tem uma prioridade, que pode ser baixa, média ou alta. Você precisa implementar uma fila para gerenciar essas solicitações com base em sua prioridade.

Escreva um programa em uma linguagem de programação de sua escolha que implemente uma fila de solicitações com as seguintes operações:

1. Adiciona uma solicitação à fila com a prioridade especificada (baixa, média ou alta).
2. Remove a próxima solicitação da fila com a maior prioridade.
3. Retorna o número de solicitações na fila.

Você pode implementar a fila usando uma lista simplesmente encadeada, por exemplo. Para testar seu programa, crie algumas solicitações de exemplo e insira-as na fila com prioridades diferentes. Em seguida, remova as solicitações da fila e verifique se elas são retornadas na ordem correta com base em sua prioridade. Por exemplo, se houver solicitações de alta prioridade na fila, elas devem ser atendidas antes das solicitações de média e baixa prioridade.

A seguir, trazemos uma sugestão de resolução utilizando português estruturado:

registro Request  
    prioridade: caractere  
    dados: caractere  
fim\_registro  
registro Node  
    solicitacao: Request  
    proximo: ponteiro  
fim\_registro  
registro TQueue  
    frente: ponteiro  
    tras: ponteiro  
    contagem: inteiro  
fim\_registro  
funcao InicializarFila(var fila: TQueue)  
    fila.frente := nulo  
    fila.tras := nulo  
    fila.contagem := 0  
fim\_funcao  
funcao Enfileirar(var fila: TQueue, prioridade, dados: caractere)  
    var  
        req: Request  
        novoNo, atual, anterior: ponteiro  
    req.prioridade := prioridade  
    req.dados := dados  
    novoNo <- novo Node  
    novoNo.solicitacao <- req  
    novoNo.proximo <- nulo  
    se (fila.frente = nulo) entao  
        fila.frente := novoNo  
        fila.tras := novoNo  
    senao  
        atual := fila.frente  
        anterior := nulo  
        enquanto (atual <> nulo) e (atual.solicitacao.prioridade <= prioridade) faca  
            anterior := atual  
            atual := atual.proximo  
        fim\_enquanto  
        se (anterior = nulo) entao  
            novoNo.proximo := fila.frente  
            fila.frente := novoNo  
        senao se (atual = nulo) entao  
            fila.tras.proximo := novoNo  
            fila.tras := novoNo  
        senao  
            anterior.proximo := novoNo  
            novoNo.proximo := atual  
        fim\_se  
    fim\_se  
    fila.contagem <- fila.contagem + 1  
fim\_funcao  
funcao Desenfileirar(var fila: TQueue) retorna Request  
    var  
        req: Request  
        temp: ponteiro  
    se (fila.frente = nulo) entao  
        escreva("A fila esta vazia.")  
    senao  
        temp := fila.frente  
        req := temp.solicitacao  
        fila.frente <- fila.frente.proximo  
        libere(temp)  
        fila.contagem <- fila.contagem - 1  
    fim\_se  
    retorne req  
fim\_funcao  
funcao TamanhoDaFila(var fila: TQueue) retorna inteiro  
    retorne fila.contagem  
fim\_funcao  
var  
    fila: TQueue  
    req: Request  
Inicio  
    InicializarFila(fila)  
    Enfileirar(fila, "alta", "Solicitacao 1")  
    Enfileirar(fila, "media", "Solicitacao 2")  
    Enfileirar(fila, "baixa", "Solicitacao 3")  
    Enfileirar(fila, "alta", "Solicitacao 4")  
    Enfileirar(fila, "media", "Solicitacao 5")  
    Enfileirar(fila, "baixa", "Solicitacao 6")  
    escreva("Numero de solicitacoes na fila: ", TamanhoDaFila(fila))  
    req <- Desenfileirar(fila)  
    se (req.prioridade <> "") entao  
        escreva("Removida solicitacao com prioridade ", req.prioridade, " e dados: ", req.dados)  
    fim\_se  
    req <- Desenfileirar

***2.3 Pilhas***

Ao contrário das filas, as pilhas são estruturas que não preservam a ordem de chegada, isto é, o elemento que sai será sempre o último a chegar, consistindo numa estratégia chamada LIFO (last in, first out). Apenas o elemento do topo da pilha poderá ser manipulado. O jogo denominado Torre de Hanói pode ser modelado como uma pilha e as peças só podem ser retiradas de cima e colocadas em cima das outras, formando pilhas. O empilhamento de caixas e contêineres também pode ser considerado exemplo.

As intervenções de inserção e remoção são realizadas apenas no topo (final) da fila. A inserção de um elemento torna-o primeiro da pilha. As operações básicas em uma pilha são:

* Empilhar(x): insere o item x no topo da pilha.
* Desempilhar(): retorna e remove o item do topo da pilha.
* Inicializa(): cria uma pilha s vazia.
* Cheia(), Vazia(): testa a pilha para saber se ela está cheia ou vazia.

Assim como as filas, nas pilhas não temos métodos para percorrer os elementos. Trabalhar com as operações básicas acima possibilita a elaboração de uma pilha genérica, permitindo sua reutilização sem a preocupação com detalhes de implementação.

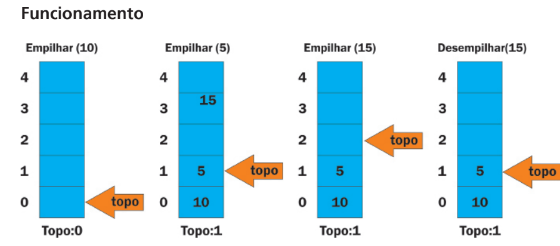


Figura 1 - Funcionamento de uma pilha

Descrição da Imagem: temos quadrados empilhados, cada um com um número ao lado, zero, um, dois, três e quatro (de baixo para cima). Os quadrados formam várias colunas. O topo da primeira coluna é o primeiro quadrado de baixo, foram empilhados quinze. Na segunda coluna, foram empilhados cinco. O número quinze está no quarto quadradinho de baixo para cima. O número cinco está localizado no segundo quadrado e o dez está no primeiro quadrado. O topo da segunda coluna é o segundo quadrado de baixo para cima. No terceiro quadrado, temos o número cinco no segundo quadrado de baixo para cima e o dez no primeiro quadrado. O topo da terceira coluna é o terceiro quadrado de baixo para cima. Na última coluna, temos o dez no primeiro quadrado de baixo para cima e o cinco no segundo quadrado. O topo da última coluna é o segundo quadrado de baixo para cima. A primeira coluna tem topo zero. A segunda, terceira e quarta coluna têm topo um.

***Exemplo de uma pilha na prática***

Uma boa maneira de entender o funcionamento de uma pilha é pensar em uma pilha de livros. Imagine que você tem uma mesa e quer empilhar alguns livros em cima dela.

Ao adicionar um livro à pilha, você o coloca no topo, empilhando-o em cima dos outros livros. Se você quiser remover um livro da pilha, você precisará retirar o livro do topo primeiro - não é possível remover um livro que esteja no meio da pilha sem retirar os livros que estão acima dele.

Essa é a essência da estrutura de dados de pilha - ela é uma coleção de elementos organizados de forma que o último elemento adicionado seja o primeiro a ser removido (conhecido como "Last In, First Out" ou LIFO em inglês).

Por exemplo, se você empilhar os livros "Harry Potter", "Senhor dos Anéis" e "Código Da Vinci", o "Código Da Vinci" estará no topo da pilha. Se você quiser retirar um livro da pilha, precisará remover o "Código Da Vinci" primeiro, seguido pelo "Senhor dos Anéis" e, por fim, pelo "Harry Potter".

Assim como na pilha de livros, em uma pilha de dados, você só pode adicionar e remover elementos no topo da pilha. Além disso, você não pode acessar elementos que estejam no meio da pilha sem antes retirar os elementos que estão acima dele.

***Implementação de uma pilha***

A implementação mais simples utiliza um vetor e uma variável para controlar o topo da pilha. A implementação utilizando uma lista ligada através de ponteiros é mais adequada, não necessitando controlar o tamanho da pilha.

Inicialmente criamos uma Classe No, que representa cada elemento da pilha.

public class No {

ver implementação no item 4.1.1

}

A classe pilha possuirá os três atributos principais das listas e aqui vamos alterar sua nomenclatura: base (indica o início da pilha, declarado com o tipo No), topo (indica o fim da pilha, declarado com o tipo No) e nomeDaPilha.

public class Pilha {  
 private No base;  
 private No topo;  
 private String nomeDaPilha;

//Construtor da classe Pilha  
 public Pilha(String nome) {  
 nomeDaPilha = nome;  
 //Como a pilha inicialmente é vazia, tanto a base como o topo  
 // receberão o valor nulo.  
 base = topo = null;  
 }

public Pilha() {  
 nomeDaPilha = "Pilha Teste";  
 base = topo = null;  
 }  
 }

Não temos um método para inserir elementos na frente, apenas o método Empilha que acrescentará os elementos no topo (final) da pilha.

public void Empilha(String item) {  
 if (Vazia())  
 base = topo = new No(item);  
 else {  
 topo.setProx(new No(item));  
 topo = topo.getProx();  
 }  
 }

Aqui também não temos um método para retirar elementos da frente, apenas o método Desempilha que removerá o topo (último elemento).

public String Desempilha() {  
 if (Vazia()) {  
 return null;  
 }  
 String item = topo.getDado();  
 if (base == topo)  
 base = topo = null;  
 else {  
 No atual = base;  
 while (atual.getProx() != topo)  
 atual = atual.getProx();  
 topo = atual;  
 atual.setProx(null);  
 }  
 return item;  
 }

O Método Vazia retorna verdadeiro se a fila está vazia e falso caso contrário.

public boolean Vazia() {  
 //Se a variável primeiro é nula, então a fila não possui elementos (nós), ela está vazia.  
 return base == null;  
 }

Teste da classe

public static void main(String[] args) {  
 Pilha p1 = new Pilha();  
 p1.Empilha("el01");  
 p1.Empilha("el02");  
 p1.Empilha("el03");  
 p1.Empilha("el04");  
 System.out.print(p1.Desempilha());  
 System.out.print(p1.Desempilha());  
 }

Saída: el04 el03

Constantemente nos deparamos com estruturas semelhantes às filas e pilhas, em bancos, repartições, restaurantes etc. Estamos sempre nos envolvendo com suas regras e definições. Vimos nesta aula que existem estruturas de dados que são apropriadas para representar as pilhas e filas, utilizando os conceitos de listas ligadas. As filas devem obedecer à estrutura FIFO, isto é, o primeiro a entrar deve ser o primeiro a sair. Já as pilhas obedecem à estrutura LIFO, isto é, o último a entrar deverá ser o primeiro a sair.

***2.3.1 Exercício sobre pilhas***

Escreva um programa que receba uma palavra e verifique se ela é um palíndromo, ou seja, se ela é a mesma palavra lida de trás para frente. Utilize uma pilha para inverter a palavra.

A seguir, trazemos uma sugestão de resolução utilizando português estruturado. Note que o programa começa declarando as variáveis que serão utilizadas: palavra e palavra\_invertida para armazenar a palavra digitada e a palavra invertida, pilha para armazenar as letras da palavra e topo para indicar o topo da pilha. Em seguida, o programa inicia o laço para percorrer a palavra digitada e ir empilhando as letras na pilha. Depois disso, o programa inicia o laço enquanto para desempilhar as letras da pilha e formar a palavra invertida. Finalmente, o programa verifica se a palavra original é igual à invertida e exibe uma mensagem indicando se a palavra é um palíndromo ou não.

algoritmo palindromo

var

palavra, palavra\_invertida: caractere

pilha: vetor[1..50] de caractere

topo: inteiro

inicio

topo <- 0

escreva("Digite uma palavra: ")

leia(palavra)

// Empilha as letras da palavra

para i de 1 ate tamanho(palavra) faca

topo <- topo + 1

pilha[topo] <- palavra[i]

fimpara

// Desempilha as letras e forma a palavra invertida

palavra\_invertida <- ""

enquanto topo > 0 faca

palavra\_invertida <- palavra\_invertida + pilha[topo]

topo <- topo - 1

fimenquanto

// Verifica se a palavra original é igual à invertida

se palavra = palavra\_invertida entao

escreva("A palavra é um palíndromo.")

senao

escreva("A palavra não é um palíndromo.")

fimse

fimalgoritmo

***2.4 Classificação: algoritmos de pesquisa***

Você certamente já realizou algum tipo de pesquisa, em que tentou encontrar se algum elemento satisfazia a condição desejada. Vamos, agora, considerar o seguinte problema: você tem que encontrar um nome em uma lista telefônica. Sair folheando as páginas indiscriminadamente seria uma atitude pouco eficaz. De forma instintiva, procuramos o nome abrindo a lista aproximadamente na metade, um pouco mais, ou pouco menos, de acordo com a inicial do nome. Se os nomes que aparecem têm suas iniciais maiores que a do nome que estamos procurando, imediatamente dividimos as folhas da esquerda e, caso contrário, dividimos as folhas da direita. Essa operação de dividir a lista é realizada até que nos tenhamos aproximado do nome desejado e, assim, podemos folhear as páginas para encontrá-lo. Esse é um método muito eficaz para realizar uma pesquisa, mas, para isso, as informações devem estar ordenadas.

Procurar informações em um conjunto desordenado demanda uma operação pouco produtiva, pois deve-se “olhar” cada um dos elementos até encontrar o desejado e, caso ele não exista, você terá consultado todos.

Pesquisar é uma atividade com a finalidade de encontrar se existe ou não elementos em um conjunto. Nesta aula, serão apresentados alguns algoritmos de pesquisa em um conjunto de dados ordenados, quais sejam: pesquisa sequencial, pesquisa binária e transformação de chave. A eficiência de cada uma está na posição em que o dado pesquisado se encontrar, o que poderá ser avaliado após conhecer os algoritmos.

***Pesquisa***

Pesquisar dados envolve determinar se um valor (denominado chave de pesquisa) está presente no conjunto de dados estudado, e, se estiver, encontrar a localização deste valor.

A forma mais simples e também a mais onerosa é a pesquisa exaustiva. Nesta técnica, o algoritmo testa cada elemento até encontrar o desejado e, quando alcança o fim do array, informa ao usuário que a chave não está presente. Esta pesquisa é geralmente utilizada em arrays desordenados e a parada do algoritmo ocorre apenas quando o elemento é encontrado ou até que todos sejam pesquisados.

Quando dispomos de um conjunto de dados ordenados, podemos utilizar algoritmos de pesquisas mais eficientes, os quais utilizam estratégias que não necessitam percorrer todos os elementos para encontrar o valor pesquisado. Além disso, não é necessário percorrer todos os elementos quando o valor não é encontrado, pois normalmente os algoritmos identificam sua ausência e param o processo.

***Pesquisa Sequencial***

Como informado anteriormente, neste algoritmo devemos considerar que o vetor está ordenado.

A busca se inicia no primeiro elemento e deve testar os elementos sequentes, devendo continuar até encontrar a chave procurada ou até encontrar o primeiro elemento com valor maior que a chave pesquisada. Como o vetor está ordenado, ao encontrar um elemento maior que a chave procurada, não é necessário continuar verificando, pois, a partir deste elemento, todos os elementos seguintes serão maiores.

Considere o vetor de dados abaixo e a chave de pesquisa 5. A pesquisa se inicia no primeiro elemento:

Quadro 1 - Vetor de Dados

| **1** | 2 | 5 | 7 | 9 | 11 | 12 | 18 | 19 | 21 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Caso não encontre, é verificado o seguinte:

Quadro 2 - Verificando os Elementos

| 1 | **2** | 5 | 7 | 9 | 11 | 12 | 18 | 19 | 21 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Os elementos seguintes são pesquisados até se encontrar o valor desejado:

Quadro 3 - Pesquisando até encontrar o Valor Desejado

| 1 | 2 | **5** | 7 | 9 | 11 | 12 | 18 | 19 | 21 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Considere agora a chave de pesquisa 8:

Quadro 4 - Considerando a chave de Pesquisa 8

| 1 | 2 | 5 | 7 | **9** | 11 | 12 | 18 | 19 | 21 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Os elementos serão visitados até se chegar ao valor 9. Como ele é maior que a chave de pesquisa (8), o algoritmo é interrompido. Temos a certeza de que ele não estará no restante do vetor, pois os demais serão, também, maiores que ele.

***Implementação da pesquisa sequencial***

O método abaixo retornará o valor da posição do elemento com o valor pesquisado ou a posição de um elemento com o valor maior. O usuário deverá verificar se a posição retornada possui o valor pesquisado (encontrou) ou se possui um valor maior que o pesquisado (não encontrou).

O algoritmo “buscaLinear” recebe (parâmetro) a chave de pesquisa e inicia um comando de repetição (while) com condição de parada: até o final do vetor ou até a chave ser maior que o elemento consultado.

public int buscaLinear(int num) {  
 int i = 0;  
 //O comando While é interrompido quando:  
 //O contador “i” é maior que o tamanho do vetor  
 ou  
 //A chave de pesquisa (num) é maior que o elemento do vetor. Neste  
 //caso, deve-se verificar se a chave encontrada corresponde à chave  
 //de pesquisa.  
 while ((i <= vetor.lenght) && (num > vetor[i]))  
 i++;  
 return i;  
 }

***Pesquisa binária***

Este algoritmo segue a estratégia de dividir para conquistar. Deve-se encontrar o elemento central do array e, a partir daí, compará-lo com a chave de pesquisa. Três situações podem ocorrer:

* O elemento central possui o valor pesquisado, encerra-se o algoritmo;
* Ele é maior que o valor pesquisado e, então, deve-se descartar os elementos a sua esquerda;
* Ele é menor que o valor pesquisado então, devem-se descartar os elementos à sua direita.

Se a chave não foi encontrada, deve-se repetir o processo, com o subconjunto de dados selecionados (esquerda ou direita), até encontrá-la ou o subconjunto ser reduzido até se confirmar que o elemento não existe e, neste caso, deve-se encerrar o algoritmo.

Considere o vetor de dados abaixo e a chave de pesquisa 5. A pesquisa se inicia no elemento central, o qual é encontrado dividindo-se o tamanho do vetor (considerar a soma dos índices: 0 início e 9 fim) por 2. O resultado deverá considerar apenas a parte inteira. Neste caso temos: (0+9) / 2 = 4,5, usar 4:

Quadro 5 - Vetor de Dados

| 0 | 1 | 2 | 3 | **4** | 5 | 6 | 7 | 8 | 9 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 5 | 7 | **9** | 11 | 12 | 18 | 19 | 21 |

Como o valor encontrado é maior que a chave de pesquisa, descartamos a parte direita do vetor e, em seguida, realizamos uma nova divisão. Cálculo do meio: (0+4) / 2 = 2:

Quadro 6 - Descarte da Parte Direita do Vetor

| 0 | 1 | **2** | 3 | 4 |
| --- | --- | --- | --- | --- |
| 1 | 2 | **5** | 7 | 9 |

E, finalmente, encontramos o valor pesquisado.

Implementação da pesquisa binária

A primeira iteração testa o elemento do meio do array.

* Se isso corresponder a chave de pesquisa, o algoritmo termina.
* Se a chave de pesquisa for menor que o elemento do meio, a chave de pesquisa não poderá localizar nenhum elemento na segunda metade do array e o algoritmo continua apenas na primeira metade.
* Se a chave de pesquisa for maior que o elemento do meio, a chave de pesquisa não poderá localizar nenhum elemento na primeira metade do array e o algoritmo continua apenas com a segunda metade do array.

Repare que a cada iteração, metade do array é descartada, ficando apenas a metade que interessa. E assim sucessivamente até localizar o elemento ou reduzindo o subarray ao tamanho zero.

O algoritmo retornará um índice (posição de um elemento no vetor) que corresponderá ao valor pesquisado ou um índice com um valor diferente ao pesquisado. O usuário deverá realizar a comparação para verificar se foi encontrado.

public int buscaBinaria(int num) {  
 int meio, inicio, fim;  
 inicio = 0;  
 fim = vetor.length - 1;  
 //Calcula o elemento central  
 meio = (int)(inicio + fim) / 2;  
 //O comando While termina quando:  
 //início maior ou igual ao fim ou  
 //a chave de pesquisa (num) igual ao valor central

while ((inicio < fim) && (num != vetor[meio])) {  
 //Se chave de pesquisa for menor que o valor central selecionam-se os  
 // elementos da esquerda.  
 if (num < vetor[meio])  
 fim = meio;  
 //Caso contrário, selecionam-se os elementos da direita  
 else  
 inicio = meio + 1;  
 meio = (int)(inicio + fim) / 2;  
 }  
 if (num == vetor[meio])  
 return meio;  
 else  
 return meio + 1;  
 }

Obter a informação desejada no menor tempo possível é diferencial para a tomada de decisões, além de ser um dos elementos principais na eficiência dos aplicativos, o qual vai ser mais “rápido” se obtiver os dados necessários em menor tempo. Vimos nesta aula que, se os dados não estiverem ordenados de alguma maneira, não temos alternativas de pesquisa e temos que realizar uma pesquisa em cada um dos elementos para verificar sua existência ou ausência. Foram, também, apresentados alguns algoritmos de pesquisa em conjunto de dados ordenados. A pesquisa sequencial visita os elementos, a partir do primeiro, um após o outro até encontrar a chave desejada, ou, se encontrar um valor maior que o pesquisado, interrompe a procura. A pesquisa binária divide o conjunto de dados em duas partes e continua procurando pela chave na parte que potencialmente ela existiria. É eleita a metade da esquerda se a chave for menor que o elemento central, caso contrário, a metade direita é escolhida. Outro algoritmo visto é a transformação de chave que possui dois métodos, tabela de endereçamento direto, que usa a chave para definir a posição na tabela de armazenamento e tabela hash que usa uma função atribuída à chave para calcular a posição de armazenamento do elemento.

***2.5 Classificação: algoritmos de ordenação***

Para começarmos, vamos pensar: estamos sempre realizando classificação das coisas. Quem nunca se deparou com uma tabela de classificação de um campeonato esportivo? A classificação no vestibular, no concurso do governo, no concurso de beleza, sempre é realizada através de parâmetros com os quais desejamos especificar uma ordem. O maior número de pontos, maior número de vitórias, quem é mais bonita (se bem que é subjetiva), o mais velho, são utilizados para definir quem será o primeiro, segundo e assim por diante.

Serão apresentados nesta aula alguns dos principais tipos existentes de algoritmos de classificação (ordenação) de dados. Entre eles serão vistos os métodos bolha, que ordena os dados fazendo com que os valores menores subam para o início do array. O método seleção direta encontra o menor valor e o posiciona na frente, fazendo o mesmo com os demais. A inserção direta compara os dois primeiros elementos ordenando-os e, em seguida, verifica o terceiro e insere na posição correta entre os dois primeiros, fazendo esse processo sucessivamente com os demais. Já o quicksort é o método mais rápido de ordenação e nele se elegem um elemento do conjunto e todos os menores que são dispostos à sua direita e todos os maiores à sua esquerda. Esse processo é repetido para cada subconjunto sucessivamente até que o vetor de dados esteja ordenado.

***Classificação***

A classificação (ordenação) é um dos processos mais utilizados na computação.

A classificação compreende um processo de rearranjar um conjunto de dados de acordo com certa relação de ordens dentre as quais podemos citar:

* Ordem alfabética;
* Ordem numérica; e
* Ordem cronológica.

Os tipos de ordenação acima podem ser dispostos de duas maneiras: ordem crescente, que rearranja os elementos do menor para o maior, e ordem decrescente, que rearranja os elementos do maior para o menor. Aqui, iremos considerar a ordem crescente para as implementações realizadas.

A classificação pode ser analisada sobre dois aspectos, com base na localização dos dados:

* **Interna:** todos os dados estão contidos na memória principal; e
* **Externa:** utilização de memória secundária.

Para fins de estudo, consideramos apenas o tipo interno de classificação.

Classificar dados, isto é, colocar os dados em ordem crescente ou decrescente, é essencial para a eficiência de algumas aplicações. Todos os algoritmos aqui estudados terão o mesmo resultado final: o vetor de dados classificado. A escolha do algoritmo afetará, principalmente, o tempo de execução e o uso de memória do programa. Os algoritmos da bolha, da seleção e da inserção (direta e binária) são simples de programar, porém não são tão eficientes. O método de classificação quicksort é o mais eficiente, mas também, é mais complicado de se implementar.

Para suporte e posterior testes da implementação dos algoritmos, vamos ver uma forma de criar um vetor com valores aleatórios.

Implementação de um vetor com números aleatórios

public void geraAleatorios() {  
 //O comando “for” irá percorrer todos as posições do vetor  
 for (int i = 0; i < valores.length; i++) {  
 //Cada posição “i” do vetor receberá um número aleatório entre 0 e 99  
 //O método random() da classe Math gera um valor aleatório entre 0 e 1  
 //0 (inclusivo) e 1 (exclusivo).  
 //Multiplicamos por 100 para termos valores entre 0 e 99  
 //Fazemos um cast (int) para obtermos um valor inteiro  
 int num = (int)(Math.random()\*100);  
 valores[i] = num;  
 }  
 }

***Bolha***

O algoritmo da bolha utiliza a técnica chamada de ordenação por submersão (sinking sort) e funciona da seguinte forma: os valores maiores submergem para o final do array ou podemos também considerar que os valores menores sobem gradualmente, até o topo do array (isto é, em direção ao início do array).

A ordenação pela submersão faz várias passagens pelo array. Cada passagem compara sucessivos pares de elementos. Se um par está em ordem crescente, os valores permanecem na mesma ordem. Se um par está em ordem decrescente, é realizada a troca entre os elementos do par.

Em cada passagem, o elemento maior será “levado” até o final do array e, assim, iremos percorrer na primeira passagem todos os elementos. Na segunda passagem, não necessitamos percorrer todos, pois o maior estará no final e, então, percorremos até o penúltimo e assim sucessivamente até que a quantidade de elementos a percorrer seja igual a 1. Temos, então, o vetor ordenado.

Quadro 1 - Vetor Desordenado

| 3 | 2 | 1 | 5 | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

1º Passagem: compara os dois primeiros elementos e ordena-os:

Quadro 2 - Comparando os Primeiros Elementos

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2** | **3** | 1 | 5 | 6 | 4 |

Na sequência compara o segundo com o terceiro e ordena-os:

Quadro 3 - Comparando o Segundo e Terceiro Elementos

|  |
| --- |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | **1** | **3** | 5 | 6 | 4 |

Continuando, compara o terceiro com o quarto e ordena-os:

Quadro 4 - Comparando o Terceiro e Quarto Elementos

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 1 | **3** | **5** | 6 | 4 |

O quarto com o quinto:

Quadro 5 - O Quarto com o Quinto

|  |
| --- |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 1 | 3 | **5** | **6** | 4 |

O quinto com o sexto (último):

Quadro 6 - Comparando os Últimos Elementos

|  |
| --- |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 1 | 3 | 5 | **4** | **6** |

Repare que o vetor já está “um pouco mais ordenado” e o maior elemento ocupa a última posição.

Repetimos o processo do primeiro ao quinto elemento e podemos verificar que o valor 5 ocupará a quinta posição. Assim, sucessivamente, vamos realizando esses passos até que a quantidade de elementos a percorrer no vetor seja igual a 1 e teremos o vetor ordenado.

Quadro 7 - Vetor Ordenado

| 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- |

Implementação da Bolha

public void Bolha() {  
 int i, aux;  
 int tam = valores.length;  
 while (tam > 1) {  
 for (i = 0; i < tam - 1; i++)  
 if (valores[i] > valores[i + 1]) {  
 aux = valores[i];  
 valores[i] = valores[i + 1];  
 valores[i + 1] = aux;  
 }  
 tam = tam - 1;  
 }  
 }

***Seleção Direta***

Tal como o algoritmo anterior, a seleção direta percorre várias vezes o vetor. A primeira iteração do algoritmo seleciona o menor elemento e o permuta pelo primeiro elemento. A segunda iteração seleciona o segundo menor elemento (que é o menor item dos elementos restantes) e o permuta pelo segundo elemento. E, assim, sucessivamente até o penúltimo elemento.

Cada iteração percorre um elemento a menos no vetor, isto é, na primeira todos os elementos são visitados, na segunda, visitamos a partir do segundo elemento, pois o primeiro irá conter o menor elemento. Assim, sucessivamente, vamos percorrendo os trechos menores do vetor, até chegarmos aos dois últimos elementos. Quando encontramos o menor entre os dois últimos, o vetor estará ordenado.

Quadro 8 - Vetor Desordenado

| 3 | 1 | 2 | 5 | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

1º Passagem: encontra o menor valor e troca com o primeiro elemento:

Quadro 9 - Encontrando menor Valor e Trocando com o Primeiro Elemento

| **1** | **3** | 2 | 5 | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

Na segunda passagem, verifica o menor a partir do segundo elemento:

Quadro 10 - Verificando o Menor

| 1 | **2** | **3** | 5 | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

Na terceira passagem, verificamos a partir do terceiro elemento e, neste caso, o menor ocupa a posição correta:

Quadro 11 - Verificando a partir do Terceiro Elemento

| 1 | 2 | **3** | 5 | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

Na quarta passagem:

Quadro 12 - Quarta Passagem

| 1 | 2 | 3 | **4** | 6 | **5** |
| --- | --- | --- | --- | --- | --- |

Na quinta e última teremos o vetor ordenado:

Quadro 13 - Vetor Ordenado

| 1 | 2 | 3 | 4 | **5** | **6** |
| --- | --- | --- | --- | --- | --- |

Implementação da seleção direta

public void Selecao() {  
 int i, j, menor, posmenor;  
 for (i = 0; i < valores.length - 1; i++) {  
 posmenor = i;  
 menor = valores[i];  
 for (j = i + 1; j < valores.length; j++)  
 if (valores[j] < menor) {  
 menor = valores[j];  
 posmenor = j;  
 }  
 valores[posmenor] = valores[i];  
 valores[i] = menor;  
 }  
 }

Inserção direta

O algoritmo de ordenação inserção direta é considerado o mais rápido entre os algoritmos considerados simples: bolha e seleção direta.

A principal característica da inserção direta está em ordenar conjunto de elementos a partir de um subconjunto ordenado localizado em seu início (os dois primeiros elementos) e, a cada novo passo, é acrescentado a este subconjunto mais um elemento na posição adequada para se manter a ordem, até que o último elemento do conjunto seja selecionado e inserido, concluindo a ordenação do vetor.

A primeira iteração seleciona os dois primeiro elementos do vetor e, se o segundo elemento for menor que o primeiro elemento, o permuta pelo primeiro elemento (ou seja, ordena apenas os dois primeiros elementos do array). A segunda iteração seleciona o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos de modo que todos os três elementos estejam em ordem. E, assim, sucessivamente para os demais elementos.

Neste algoritmo, ao se procurar o local ideal para inserir o novo elemento entre os elementos já ordenados, pode-se optar por uma busca sequencial ou pela busca binária.

Quadro 14 - Vetor Desordenado

| 3 | 1 | 2 | 5 | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

Inicialmente os dois primeiros elementos são ordenados:

Quadro 15 - Primeiros Elementos Ordenados

| **1** | **3** | 2 | 5 | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

Em seguida, tomamos o terceiro elemento e inserimos na posição adequada entre os dois primeiro, de modo que eles fiquem ordenados:

Quadro 16 - Ordenando Elementos

| **1** | **2** | **3** | 5 | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

No próximo passo, tomamos o quarto elemento e repetimos o procedimento anterior:

Quadro 17 - Repetindo o Procedimento Anterior

| **1** | **2** | **3** | **5** | 6 | 4 |
| --- | --- | --- | --- | --- | --- |

Continuando, tomamos o quinto elemento:

Quadro 18 - Tomando o Quinto Elemento

| **1** | **2** | **3** | **5** | **6** | 4 |
| --- | --- | --- | --- | --- | --- |

Finalmente, tomamos o último elemento:

Quadro 19 - Tomando o Último Elemento

| **1** | **2** | **3** | **4** | **5** | **6** |
| --- | --- | --- | --- | --- | --- |

Podemos perceber que ao inserirmos um elemento entre o subconjunto de elementos já ordenados, precisamos realizar o deslocamento uma casa à direita de todos os elementos que forem maiores que ele.

***Implementação inserção direta***

public void insercaoDireta() {  
 int aux, i, j, posi;  
 //Comparamos os dois primeiros elementos, se o segundo for menor,  
 // invertemos suas posições.  
 if (valores[0] > valores[1]) {  
 aux = valores[0];  
 valores[0] = valores[1];  
 valores[1] = aux;  
 }  
 for (i = 2; i < valores.length; i++) {  
 aux = valores[i];  
 //Chamamos o método buscaLinear a passamos por parâmetro o  
 // elemento a ser inserido e o tamanho subconjunto de elementos do  
 // vetor. A variável posi receberá a posição adequada para se inserir o  
 //elemento no subconjunto.  
 posi = buscaLinear(aux, i - 1);  
 //O comando “for” desloca todos os elementos, a partir da posição  
 // encontrada, uma casa a direita.  
 for (j = i; j > posi; j--)  
 valores[j] = valores[j - 1];  
 //Insere o elemento na posição encontrada. Ordenado em relação ao  
 // subconjunto.  
 valores[posi] = aux;  
 }  
 }

public int buscaLinear(int num, int tam) {

int i = 0;

//O comando While é interrompido quando:

//O contador “i” maior que o tamanho do vetor ou

//A chave de pesquisa (num) maior que o elemento do vetor

while ((i <= tam) && (num > valores [i]))

i++;

return i;

}

Os algoritmos de ordenação são de extrema importância para a eficiência de muitos aplicativos, os quais necessitam da rapidez no acesso às informações. A ordenação é caracterizada pelo rearranjo dos elementos de um array, onde uma relação de ordem é especificada: alfabética, numérica, cronológica etc. (crescente ou decrescente). Vimos que os algoritmos apresentados são relativamente simples de implementar, não necessitando de muitos comandos ou lógica elaborada. Foram apresentados quatro tipos diferentes de algoritmos, sendo os mais comuns e simples a bolha, a seleção direta e a inserção direta, os quais são mais fáceis de implementar mas não tão rápidos quando comparados ao quicksort. O quicksort é considerado como o mais veloz, em média, dos algoritmos conhecidos, mas mais complexo de se implementar, se comparado com os outros apresentados.

**3- Tipos de árvore**

***3.1 Árvores***

Como sabemos, em muitos casos, os pesquisadores utilizam estruturas da natureza como embasamento para suas descobertas. O movimento das formigas já foi objeto de estudos para algoritmos de definição de rotas (caminhos), o DNA e o processo de reprodução foram base para um dos principais métodos para cálculo do melhor (menor) caminho. As árvores são muito utilizadas na representação genealógica, pois sua estrutura de ramificações é ideal para a representação das conexões familiares e ou hierárquicas.

Na computação, existem diversas aplicações que necessitam de estruturas mais complexas que as listas estudadas até agora para disposição dos seus elementos. A representação hierárquica dos funcionários da empresa ACME pode ser associada a uma árvore invertida:

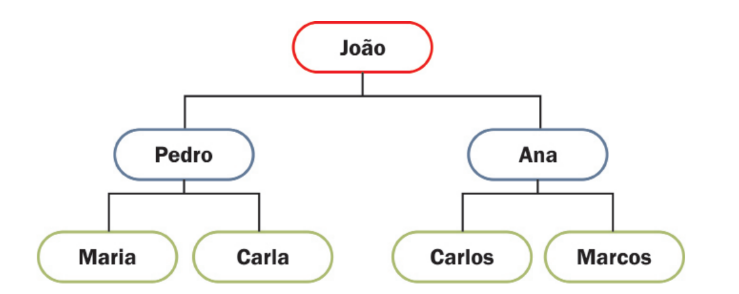


Figura 1 - Representação hierárquica de quadro funcional

Descrição da imagem: diagrama hierárquico em que o nível 1, mais alto, é ocupado por João; o segundo nível é ocupado por Pedro e Ana; e o terceiro nível é ocupado por Maria, Carla, que derivam de Pedro, e Carlos e Marcos, que derivam de Ana.

A presidência seria a raiz, os diretores o caule e a primeira ramificação e os demais cargos, seriam representados nas outras ramificações até o nível das folhas. Inúmeros problemas do cotidiano podem ser modelados através de modelos de árvores.

As árvores utilizam uma estrutura de nós, mas são conceitualmente diferentes das listas encadeadas. Nas listas, os elementos (nós) são ligados sequencialmente, já nas árvores os elementos estão dispostos de forma hierárquica. A estrutura das árvores é composta por:

* Raiz: elemento principal;
* Ramos ou filhos; e
* Folha ou nó terminal.

A raiz é ligada aos ramos ou filhos. Estes são ligados a outros elementos que também possuem outros ramos. O elemento da extremidade, que não possui ramos, é chamado folha ou nó terminal.

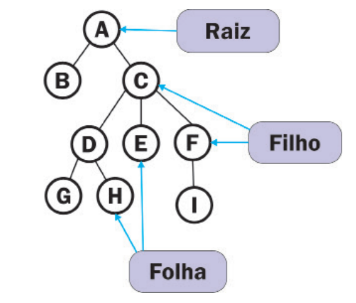


Figura 2 - Representação gráfica da raiz ligada aos ramos ou filhos

Descrição da imagem: O elemento A (Raiz) está ligado aos elementos B e C. C também está ligado a D, E e F (Filho). F está ligado a I. D está ligado a G e H (Folha).

Definições:

* Nós folhas: são aqueles que não têm filhos. Exemplo: {B, G , H, E, I}
* Ramos ou filhos: não são nós folhas, ou seja, são aqueles que possuem 1 ou mais descendentes. Ex: {A, C, D, F}

Exemplo de representação em árvore: expressão aritmética a + (b \* (c / d) – e)

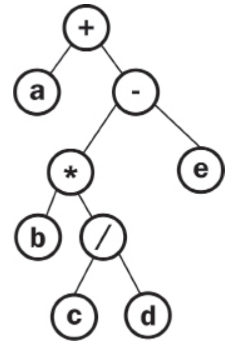


Figura 3 - Exemplo de representação em árvore

Descrição da imagem: O elemento A está ligado ao elemento mais. Mais está ligada a menos. Menos está ligado a E e vezes. Vezes está ligado a B e divisão. Divisão está ligada a C e D.

 O Nível (profundidade) e altura de um nó pode ser conferido na figura a seguir.

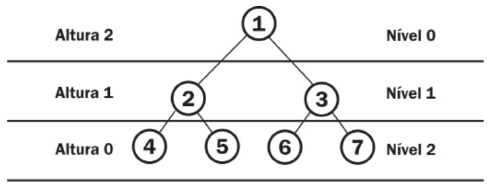


Figura 4 - Altura e profundidade de um nó

Descrição da imagem: O topo da árvore contém o elemento um e é nível zero e altura dois. No meio há os elementos dois e três que estão ligados ao um e é nível um e altura um. E bem abaixo há os elementos quatro e cinco que estão ligados ao dois e seis e sete que estão ligados ao três, a altura é zero e o nível é dois.

O nível ou profundidade de um nó “n” é o número de nós existente entre a raiz até o nó “n”. Portanto, o nível da raiz é 0.

A altura de um nó “n” é o número de nós no maior caminho de “n” até um de seus descendentes. As folhas têm altura 0.

**Grau:** de um nó é o número de filhos do nó. Da árvore, é o máximo de filhos de um nó. No exemplo acima, a árvore tem grau 2 (dois).

***3.1.1 Tipos de representações de árvores***

Uma árvore pode ser representada, implementada, com base na disposição de seus nós (elementos) na memória: **Representação por Adjacência e Representação Encadeada**.

***Representação por adjacência***

Neste tipo de representação, os nós são armazenados sequencialmente, de acordo com uma convenção pré-determinada.

Essa representação é muito útil para armazenamento permanente (discos e fitas), mas ineficiente para manipular árvores dinâmicas (necessitam de operações de inserções e remoções de nós).

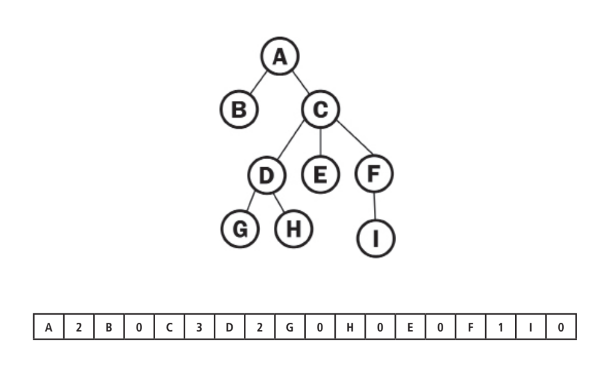


Figura 1 - Representação por Adjacência

Os valores representam a quantidade de filhos que cada nó possui. Assim, “A” possui dois filhos, os nós “B” e “C”, pois "A" está ligada a "B" e "C". “B” não possui nenhum filho e “C” possui 3 filhos: “D” (que possui 2 filhos: “G” e “H”), “E” que não possui filhos e “F” que possui 1 filho (“I”).

***Representação encadeada***

Este tipo de representação emprega os conceitos de listas encadeadas onde cada nó (elemento) contém: As informações do nó e Referências aos ramos do nó (ponteiro para cada um dos seus nós filhos).

A desvantagem desta representação é a grande quantidade de referências a subárvores nulas.

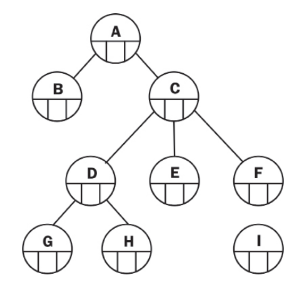


Figura 2 - Representação Encadeada

Na árvore acima, temos o nó-raiz “ligado” aos nós “B” e “C” e o nó “B” não possui filhos. Já o nó “C” possui ligação aos nós “D”, “E” e “F”. Nesta representação, deve-se definir inicialmente a quantidade máxima de filhos que cada nó pode possuir, que, neste exemplo, é 3. Assim, ao se implementar cada nó, a quantidade de ponteiros deve ser determinada.

***3.1.2 Árvores binárias***

Uma árvore binária é uma estrutura de dados bidimensional, com propriedades especiais.

O nó-raiz é o primeiro nó da árvore. Cada ligação no nó-raiz referencia um filho.

Os nós de uma árvore binária contêm, no máximo, duas ligações: o filho esquerdo e o filho direito.

O filho esquerdo é o primeiro nó na subárvore esquerda (também conhecido como o nó-raiz da subárvore esquerda). E o filho direito é o primeiro nó na subárvore direita (também conhecido como o nó-raiz da subárvore direita).

O nó sem filhos é chamado de nó-folha.

Representação gráfica de uma árvore binária:

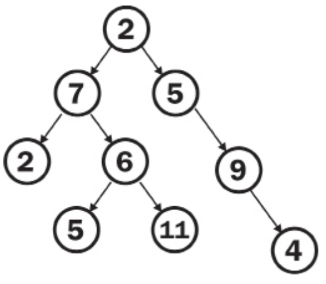


Figura 1 - Árvore Binária

Descrição da imagem: O elemento dois está apontando para o elemento cinco que aponta para o elemento nove e este aponta para o elemento quatro. Dois também está apontando para o elemento sete que de um lado aponta para o elemento dois e do outro lado aponta para o elemento seis. Seis aponta para os elementos cinco e onze.

***Exemplo prático de árvore binária***

Para entender o conceito de uma árvore binária, pense em uma árvore com galhos que se dividem em dois a cada bifurcação. Cada bifurcação cria dois ramos, um para a esquerda e outro para a direita.

Por exemplo, imagine uma árvore binária que representa as escolhas que você pode fazer em um jogo de adivinhação. O primeiro nó da árvore pergunta se você está pensando em um animal terrestre ou um animal aquático. Se você escolher "terrestre", você seguirá o ramo da esquerda, que perguntará se o animal tem quatro pernas ou não. Se você escolher "aquático", você seguirá o ramo da direita, que perguntará se o animal é um peixe ou não. Essas perguntas se bifurcam continuamente até que você chegue a uma folha da árvore, que representa a resposta final.

Assim como na árvore binária do jogo de adivinhação, uma árvore binária em ciência da computação é formada por nós que se bifurcam em dois ramos, um para a esquerda e outro para a direita. Cada nó pode ter até dois filhos (ou ramos). O primeiro nó da árvore é conhecido como a raiz da árvore. Cada nó que não tem filhos é chamado de folha.

Por exemplo, imagine que você queira criar uma árvore binária para armazenar os números 1, 2, 3 e 4. Você começaria criando o nó raiz com o valor 1. Em seguida, você criaria um nó à esquerda com o valor 2 e um nó à direita com o valor 3. Por fim, você adicionaria um nó à esquerda do nó com o valor 2 com o valor 4.

Assim, a árvore binária é formada por uma hierarquia de nós, onde cada nó tem um valor e pode ter até dois filhos, um à esquerda e outro à direita. A árvore pode ser percorrida em pré-ordem, em ordem ou pós-ordem, e é uma das estruturas de dados mais utilizadas em ciência da computação.

***Percurso em árvores binárias***

O percurso em uma árvore binária é a ordem com que todos os seus nós são visitados. Vamos considerar os seguintes percursos:

* Pré-ordem;
* In-ordem;
* Pós-ordem.

Os percursos em árvores utilizam a técnica de recursividade nos algoritmos. A recursividade é originada quando uma função chama a si própria.

***Travessia em pré-ordem***

Os seguintes passos devem ser executados para se percorrer os nós da árvore:

* Se árvore vazia, fim;
* Exibir a informação do nó;
* Percorrer em pré-ordem a subárvore esquerda (recursivamente);
* Percorrer em pré-ordem a subárvore direita (recursivamente).



Figura 2 - Travessia em pré-ordem

Descrição da imagem: No topo há o elemento dois, chamado de primeiro. Ele está ligado ao um (segundo) e oito (terceiro). Oito está ligado ao quatro (quarto) e nove (nono). Quatro está ligado a três (quinto) e seis (sexto). Seis está ligado ao cinco (sétimo) e sete (oitavo). O resultado da In-Ordem é dois, um, oito, quatro, três, seis, cinco, sete e nove.

***Percorrendo a árvore acima em Pré-Ordem***

Escreve(2);  
PreOrdem(2, esq);  
Escreve(1);  
PreOrdem(1, esq);  
PreOrdem(1, dir);  
PreOrdem(2, dir);  
Escreve(8);  
PreOrdem(8, esq);  
Escreve(4);  
PreOrdem(4, esq);  
Escreve(3);  
PreOrdem(3, esq);  
PreOrdem(3, dir);  
PreOrdem(4, dir)  
Escreve(6);  
PreOrdem(6, esq);  
Escreve(5);  
PreOrdem(5, esq);  
PreOrdem(5, dir);  
PreOrdem(6, dir);  
Escreve(7);  
PreOrdem(7, esq);  
PreOrdem(7, dir);  
PreOrdem(8, dir);  
Escreve(9);  
PreOrdem(9, esq);  
PreOrdem(9, dir);

***Travessia em In-Ordem***

* Se árvore vazia, fim;
* Percorrer em pré-ordem a subárvore esquerda (recursivamente);
* Exibir a informação do nó;
* Percorrer em pré-ordem a subárvore direita (recursivamente).

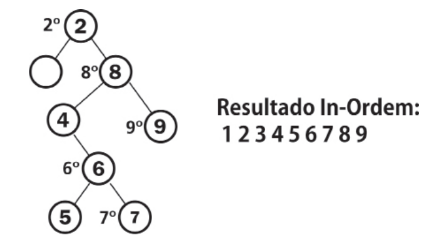


Figura 3 - Travessia In-ordem

Descrição da imagem: O elemento mais alto é o dois (segundo) e está ligado ao oito (oitavo) e a um elemento em branco. O oito está ligado ao quatro e ao nove (nono). O quatro está ligado ao elemento seis (sexto) e este está ligado aos elementos cinco e sete (sétimo). O resultado da In-Ordem é um, dois, três, quatro, cinco, seis, sete, oito e nove.

Travessia em Pós-Ordem

* Se árvore vazia, fim;
* Percorrer em pré-ordem a subárvore esquerda (recursivamente);
* Percorrer em pré-ordem a subárvore direita (recursivamente); e
* Exibir a informação do nó.

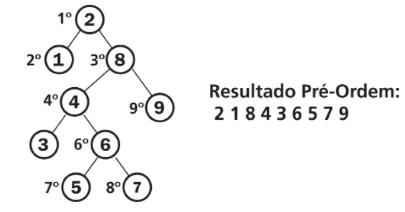


Figura 4 - Travessia em Pós-Ordem

Descrição da imagem: No topo há o elemento dois (primeiro) ligado aos elementos um (segundo) e oito (terceiro). O oito está ligado aos elementos quatro (quarto) e nove (nono). O quatro está ligado aos elementos três e seis (sexto). O seis está ligado aos elementos cinco (sétimo) e sete (oitavo). O resultado da Pré-Ordem é dois, um, oito, quatro, três, seis, cinco, sete e nove.

***3.1.3 Árvores binárias de busca (pesquisa)***

Uma importante aplicação de árvores binárias é a pesquisa. A finalidade desta árvore é estruturar os dados de tal maneira que possibilite uma pesquisa binária.

São consideradas árvores binárias de busca as estruturas que obedecem às seguintes características:

* Todas as chaves da subárvore esquerda são menores que a chave da raiz;
* Todas as chaves da subárvore direita são maiores que a chave raiz; e
* As subárvores direita e esquerda são também árvores binárias de busca.

Essa estrutura possibilita que a busca por uma determinada chave seja realizada com um número menor de comparações. A busca de uma chave “n” segue os seguintes passos:

* Inicialmente compare com a raiz;
* Se menor, vá para a subárvore esquerda; e
* Se maior, para a subárvore direita.

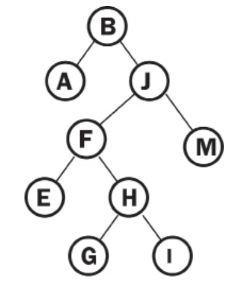


Figura 1 - Árvore Binária de Busca

Sabendo que a árvore acima é uma árvore binária de busca, para encontrarmos a chave “H” vamos percorrer os seguintes nós: B -> J -> F -> H

Assim, começando pela raiz temos: H é maior que B, segue a subárvore da direita; “H” é menor que “J”, segue a subárvore da esquerda; “H” é maior que “F”, segue a subárvore da direita e finalmente encontramos a chave.

Vamos ver abaixo as operações associadas ao TAD árvore binária de busca:

* Inicializar uma árvore binária de busca;
* Verificar se a árvore está vazia ou não;
* Inserir na árvore binária;
* Remover na árvore binária;
* Busca na árvore (com ou sem recursividade).

***Inserção em árvores binárias de busca***

As operações em árvores binárias devem satisfazer as regras que as definem, isto é, todas as chaves da subárvore esquerda devem ser menores que a chave da raiz e todas as chaves da subárvore direita devem ser maiores que a chave da raiz.

Depois de realizada uma inserção, a árvore deverá manter suas características. Para inserir um nó na árvore, são necessários os seguintes passos:

* Pesquisar a chave a ser inserida; e
* Se não achar, inserir no último nó da pesquisa.

No exemplo abaixo, a chave 5 deve ser inserida na árvore:

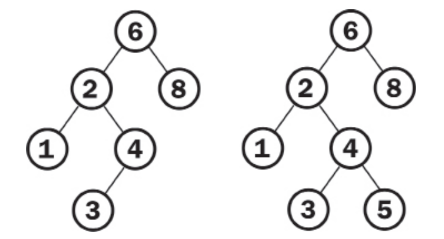


Figura 2 - Exemplo de Inserção em Árvores Binárias de Busca

Descrição da imagem: No primeiro exemplo, oito está ligado a seis, que está ligado a dois. Dois está ligado a um e quatro. Quatro está ligado a três. No segundo exemplo, oito está ligado a seis, que está ligado a dois. Dois está ligado a um e quatro. Quatro está ligado a três e cinco.

Realizamos a pesquisa da chave 5 e, inicialmente, comparamos com a raiz 6. Como o 5 é menor, vamos pela subárvore à esquerda. Comparamos com o nó 2 e, como 5 é maior, vamos pela direita, onde comparamos com o 4 e, como 5 é maior, vamos pela direita. Como o nó 4 é uma folha, não temos mais como caminhar e, então, inserimos o novo nó.

***Remoção em árvores binárias***

Para completarmos as operações, precisamos entender o processo de remoção em árvores binárias.

A operação de remoção de um nó é mais complexa e envolve alguns arranjos na árvore binária. De acordo com a posição do nó a ser removido, temos três condições que devem ser observadas:

* Se o nó é folha, apenas remover imediatamente;
* Se possuir um filho, pode ser removido após os ajustes do ponteiro de seu pai, isto é, seu pai deverá apontar para seu filho; e
* Se o nó possuir dois filhos, deve-se substituir este nó com o nó da subárvore à direita que possuir a chave com o menor valor e remover aquele nó. Como o menor nó da subárvore direita não pode ter um filho à esquerda, a sua remoção segue a condição do item “a”.

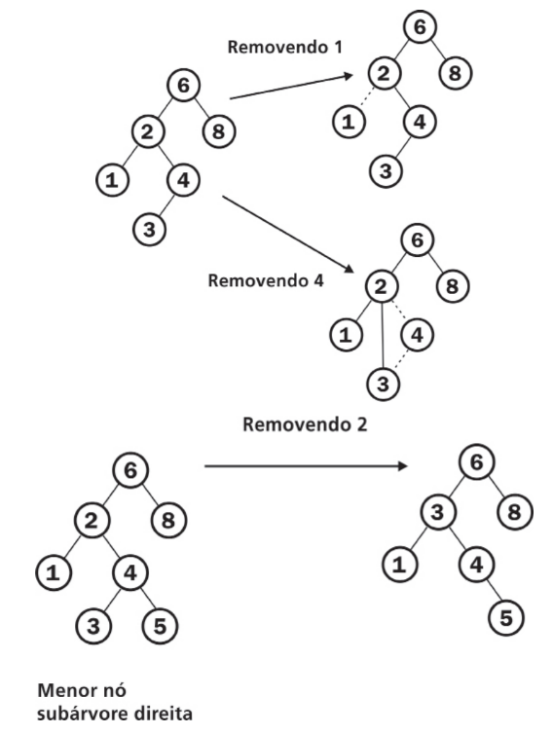


Figura 3 - Exemplo de Remoção em Árvores Binárias

Descrição da imagem: Na árvore de exemplo temos o elemento oito ligado ao seis, que está ligado ao dois. O dois está ligado ao um e ao quatro. O quatro está ligado ao três. Há uma flecha indicando a remoção de um elemento. O ligamento entre o dois e o um está demarcado. Também há uma flecha mostrando a remoção do quatro. Uma linha demarca a ligação entre o dois e o quatro, e o quatro com o três. No próximo exemplo temos o oito que está ligado ao seis, que está ligado ao dois. O dois está ligado ao um e ao quatro. O quatro está ligado ao três e ao cinco. Foi removido o menor nó da subárvore direita, ou seja, o elemento três.

**3.1.4 Árvores balanceadas**

Árvores de pesquisa devem manter uma simetria entre as subárvores da direita e esquerda de sua raiz e assim sucessivamente para cada nó. Isso possibilita uma maior eficiência na procura de uma chave. Uma árvore que possua um dos lados muito maior que o outro (desbalanceada) perde sua eficiência na pesquisa, pois, temos que percorrer muitos nós, em sequência, para encontrar uma chave posicionada nas folhas.

Uma árvore é considerada balanceada se, para cada nó, as alturas de suas subárvores diferem de, no máximo, 1.

Considere os seguintes exemplos de árvore binária:

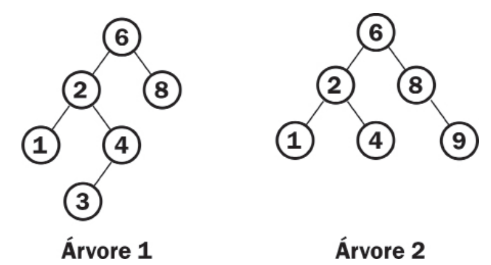


Figura 1 - Exemplos de Árvore Binária

Descrição da imagem: Na primeira árvore, o elemento seis está no topo e está ligado a oito e dois. Dois está ligado a um e quatro. Quatro está ligado a três. Na segunda árvore, o elemento seis, ainda no topo, está ligado a dois e a oito. Dois está ligado a um e a quatro, enquanto oito está ligado ao nove.

Árvore 01: está desbalanceada, pois, considerando o nó 6, a altura da subárvore esquerda é igual a 3 e a altura da subárvore direita é igual a 1, a diferença 3 – 1 = 2.

Árvore 02: está balanceada, pois todas as subárvores de cada nó têm diferença de altura em no máximo 1.

A inserção de chaves em uma árvore binária poderá provocar o seu desbalanceamento, o que pode tornar a busca tão ineficiente quanto a busca sequencial (no pior caso). A solução neste caso seria o balanceamento da árvore quando necessário.

Considerando a Árvore 02 representada acima, insira a chave 10. Teríamos o seguinte resultado:

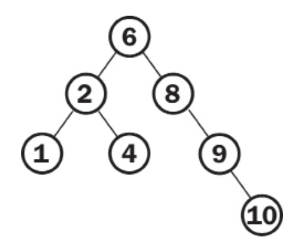


Figura 2 - Resultado da Árvore 02 com a chave 10 inserida

Teríamos como resultado uma árvore binária desbalanceada. Uma vez que a altura da subárvore à direita do nó 8 é 2 e a altura da sua subárvore à esquerda é 0, portanto, teríamos uma diferença maior que 1.

***Exemplo prático de uma árvore balanceada***

Uma maneira lúdica de entender o conceito de árvore balanceada é imaginar um jogo de Jenga, onde cada bloco representa um nó na árvore. O objetivo do jogo é remover blocos da torre sem derrubá-la. Para fazer isso, você precisa escolher com cuidado qual bloco remover e em que ordem, de forma a manter a torre equilibrada.

Por exemplo, imagine que você quer criar uma árvore balanceada para armazenar informações sobre as equipes de um campeonato de futebol. Cada nó da árvore representa uma equipe e é organizado em ordem alfabética de acordo com o nome da equipe.

Para garantir que a árvore seja balanceada, você precisa escolher um algoritmo de inserção que mantenha a altura da árvore sempre a menor possível. Um exemplo de algoritmo de inserção é o algoritmo AVL, que você verá na sequência, que verifica a diferença de altura entre as subárvores à esquerda e à direita de cada nó e faz rotações para equilibrar a árvore, se necessário.

Assim, a árvore balanceada é formada por uma hierarquia de nós, cada um representando uma equipe no campeonato de futebol e organizado em ordem alfabética. A árvore é equilibrada para garantir que a busca seja realizada de forma eficiente, mantendo a altura da árvore a menor possível. A árvore balanceada é uma das estruturas de dados mais utilizadas em bancos de dados e sistemas de arquivos, onde o desempenho é crítico para o sucesso da aplicação.

***Balanceamento de árvores binárias***

Árvores AVL foram propostas pelos matemáticos russos (G.M. **A**delson-**V**elskki e E.M.**L**andis). Após as operações de inserção e remoção, podemos gerar o desbalanceamento da árvore. Nestes casos, precisamos nos preocupar em reparar o seu balanço. A restauração deste balanço é efetuada através de operações chamadas de rotações.

Como já vimos, uma árvore é dita balanceada se a diferença entre a altura da subárvore à direita e a altura da subárvore à esquerda for no máximo 1. A esta condição damos o nome de fator de balanceamento (FB).

***Balanceamento***

1º - Calcular o fator de balanceamento de cada nó FB (nó) = (altura da subárvore à direita – altura da subárvore à esquerda)

2º - Para uma árvore ser AVL, os fatores de balanço devem ser necessariamente -1, 0, ou 1;

3º - O nó com FB >1 ou <-1 deve ser balanceado. O balanceamento de um nó é feito através de operações denominadas rotações. Substituição de um nó raiz por um de seus filhos, descendo este um nível:

**Rotação simples:** os sinais do FB do nó desbalanceado e de seu filho são iguais, isto é, ambos negativos ou positivos. Exemplo, (+2 e +1) ou (-2 e -1). Neste caso, o nó filho deve ser posicionado no lugar da raiz da subárvore. A rotação pode ser à direita, quando o nó filho à esquerda toma o lugar da raiz ou à esquerda, quando o nó filho à direita toma o lugar da raiz.

**Rotação dupla:** os sinais do FB do nó desbalanceado e de seu filho são diferentes, isto é, um positivo e outro negativo. Exemplo, (+2 e -1) ou (-2 e +1). Neste caso, devemos realizar, inicialmente, uma rotação na subárvore do nó filho e em seguida fazer uma rotação simples.

**Rotação simples**

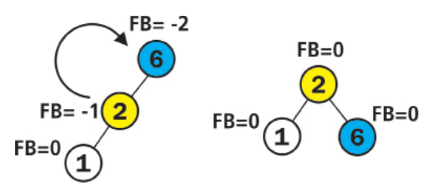


Figura 3 - Exemplo de Rotação Simples

Descrição da imagem: O elemento um (FB=zero) está ligado ao dois (FB=menos um). O dois está ligado ao seis (FB=menos dois). Há uma flecha indicando a ligação entre o dois e o seis. Ao lado, temos outra árvore, formando um triângulo. No topo está o dois (FB=zero). Ele está ligado ao um (FB=zero) e ao seis (FB=zero).

Em alguns casos as operações de rotação necessitam de remanejamento de nós para manter a característica da árvore binária de busca. Veja o exemplo abaixo:

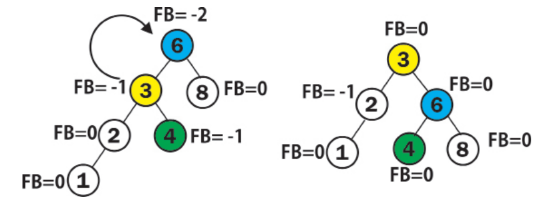


Figura 4 - Remanejamento de Rotação

Descrição da imagem: Na primeira árvore, seis está no topo (FB=menos dois). Ele está ligado ao três (FB=menos um) e ao oito (FB=0). Há uma flecha indicando a ligação entre o três e o seis. O três está ligado ao dois (FB=zero) e ao quatro (FB=menos um). O dois está ligado ao um (FB=zero). Na segunda árvore, três está no topo (FB=zero) e está ligado ao dois (FB=menos um) e ao seis (FB=zero). Dois está ligado ao um (FB=zero). Seis está ligado ao quatro (FB=zero) e ao oito (FB=zero).

Após a inserção do nó 1, o nó 6 tornou-se desbalanceado. Ao realizar a rotação do nó 3, seu filho, o nó 4, deve ser reposicionado na subárvore à direita.

Rotação dupla

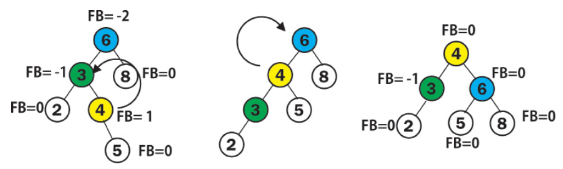


Figura 5 - Exemplo de Rotação Dupla

Descrição da imagem: Na primeira árvore, o elemento seis (FB=menos dois) está no topo e está ligado ao três (FB=menos um) e ao oito (FB=zero). O três está ligado ao dois (FB=zero) e ao quatro (FB=um). O quatro está ligado ao cinco (FB=zero). Há uma flecha apontando do elemento quatro ao elemento três. Na segunda árvore, o elemento seis está no topo e está ligado ao quatro e oito. Há uma flecha indicando do elemento quatro ao seis. O quatro está ligado ao três e ao cinco. O três está ligado ao dois. Na terceira árvore, o quatro (FB=zero), está no topo e está ligado ao três (FB=menos um) e ao seis (FB=zero). O três está ligado ao dois (FB=zero). O seis está ligado ao cinco (FB=zero) e ao oito (FB=zero).

No caso acima, o nó 6 está desbalanceado, FB(6) = -2 e o FB(3) = 1 e, como os sinais estão trocados, devemos realizar a rotação dupla. Primeiramente, realizamos a rotação simples do nó 4 com o nó 3 e fazemos os ajustes necessários e, em seguida, realizamos a rotação simples do nó 4 com o nó 6 e fazemos os ajustes necessários.

**3.1.5 Árvores B**

Árvores B (criada por Rudolf Bayer e Edward Meyers McCreight) são árvores de pesquisa balanceadas especialmente projetadas para trabalharem em memória secundária, pesquisa de informações em discos magnéticos, pois tornam mínima a quantidade de operações de movimentação de dados (escrita/leitura) numa pesquisa ou alteração.

Elas não necessitam ser balanceadas frequentemente como as árvores de busca binária. Árvores B possuem vantagens em relação a outros tipos de implementações quanto ao tempo de acesso e pesquisa aos nós.

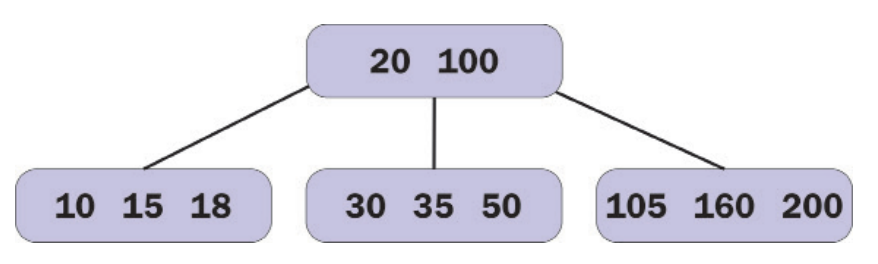


Figura 1 - Exemplo de Árvore B

Descrição da imagem: No topo da árvore há um elemento com os números vinte e cem dentro dele. Ele está ligado a três elementos. Dentro do primeiro há os números dez, quinze e dezoito. No segundo há os números trinta, trinta e cinco e cinquenta. No terceiro elemento há os números cento e cinco, cento e sessenta e duzentos.

***Estrutura do nó***

Os nós em árvores B, também conhecidos como páginas, geralmente são representados por um conjunto de elementos (chaves) apontando para seus filhos, que por sua vez também podem ser conhecidos como folhas. Cada elemento dos nós contém um ponteiro para uma subárvore com chaves menores e um ponteiro para uma subárvore com chaves maiores.



Figura 2 - Exemplo de Estrutura de um Nó

***Exemplo prático de uma árvore B***

Uma boa maneira de entender o conceito de uma árvore B é imaginar uma biblioteca com prateleiras que podem armazenar vários livros. Cada prateleira tem uma capacidade máxima de livros que pode armazenar e é organizada em ordem crescente de acordo com o número de páginas dos livros.

Por exemplo, imagine que você queira criar uma árvore B para armazenar informações sobre os funcionários de uma empresa. Cada nó da árvore B representa uma prateleira na biblioteca e tem uma capacidade máxima de funcionários que pode armazenar. Os funcionários são organizados em ordem alfabética de acordo com seus nomes.

Cada nó da árvore B tem um número mínimo e máximo de filhos. O número mínimo de filhos é conhecido como o fator de carga mínimo e é calculado a partir da capacidade mínima de cada prateleira. O número máximo de filhos é conhecido como o fator de carga máximo e é calculado a partir da capacidade máxima de cada prateleira.

Por exemplo, imagine que você defina o fator de carga mínimo como 2 e o fator de carga máximo como 4. Isso significa que cada nó da árvore B pode ter de 2 a 4 filhos, representando de 2 a 4 prateleiras na biblioteca.

Assim, a árvore B é formada por uma hierarquia de nós, cada um representando uma prateleira na biblioteca e organizada em ordem crescente. A árvore pode ser percorrida em ordem para recuperar as informações armazenadas nela. A árvore B é uma das estruturas de dados mais utilizadas em bancos de dados e sistemas de arquivos.

Estrutura Árvore B

Árvore B de ordem "m" (máximo de filhos para cada nó) é uma árvore que possui as seguintes propriedades:

* Cada nó tem no máximo "m" filhos;
* Cada nó (exceto a raiz e as folhas) tem pelo menos "m/2" filhos;
* A raiz tem pelo menos dois filhos, se ela mesma não for uma folha;
* Todas as folhas aparecem no mesmo nível e carregam informação; e
* Um nó não-folha com "k" filhos deve ter k-1 chaves.

Muitas vezes necessitamos de estruturas mais complexas que as listas para uma adequada representação dos elementos manipulados. São estruturas muito utilizadas na computação. Árvores são compostas por elementos chamados nós, os quais são dispostos em uma estrutura hierárquica: um nó raiz e suas ramificações, os filhos e um nó sem filhos, que é denominado folha. Uma de suas variações, as árvores binárias, são utilizadas para pesquisas. Com um número máximo de dois filhos por nó e uma subárvore do nó podem ser compostas por chaves menores (subárvore esquerda) e por chaves maiores (subárvore) direita, que é otimizada para uma pesquisa binária. Outra variação, árvores B, são especialmente criadas para a pesquisa em memória secundária, sendo uma adaptação das árvores binárias para se adequarem aos dispositivos magnéticos.