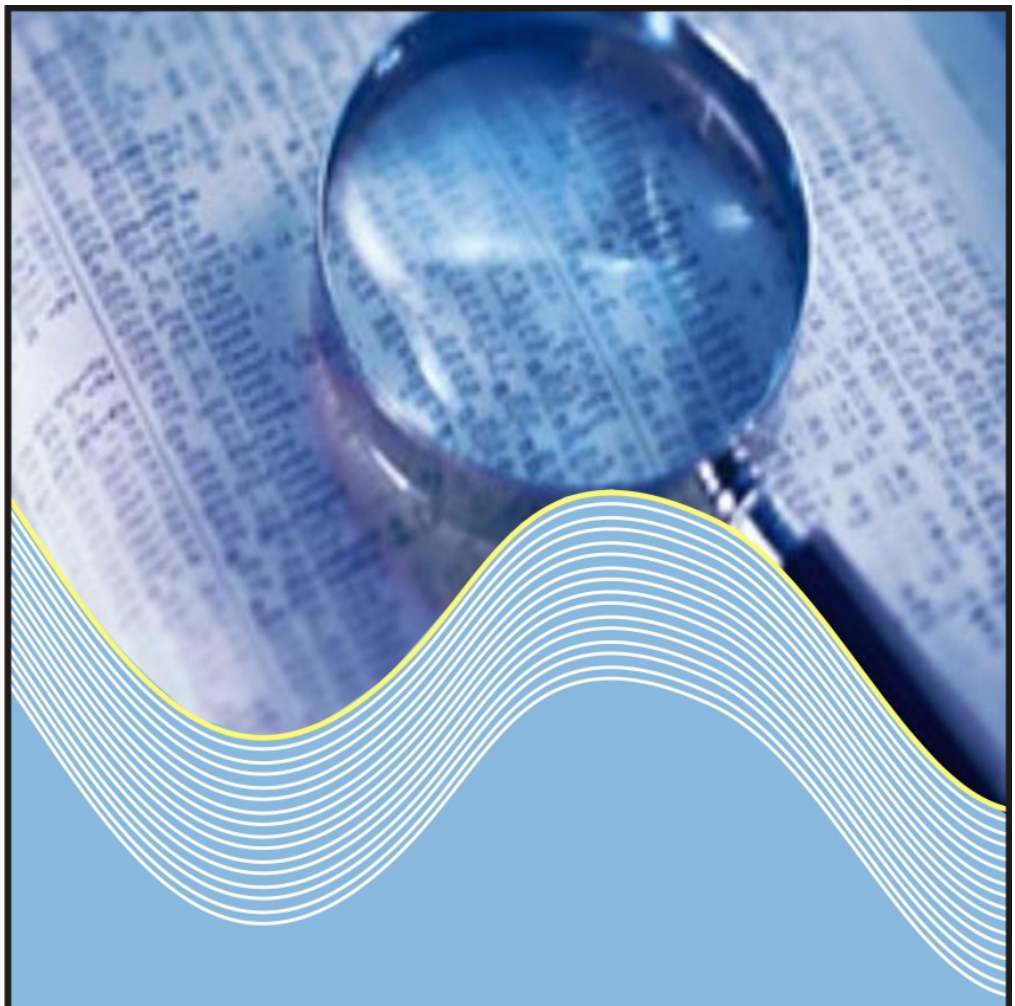


MÓDULO III

Estruturas de Dados

André Macêdo Santana
Erico Meneses Leão

FICHA BIBLIOGRÁFICA

PRESIDENTE DA REPÚBLICA
Luiz Inácio Lula da Silva

MINISTRO DA EDUCAÇÃO
Fernando Haddad

GOVERNADOR DO ESTADO DO PIAUÍ
Wellington Dias

UNIVERSIDADE FEDERAL DO PIAUÍ
Luiz de Sousa Santos Júnior

SECRETÁRIO DE EDUCAÇÃO A DISTÂNCIA DO MEC
Carlos Eduardo Bielschowsky

COORDENADORIA GERAL DA UNIVERSIDADE ABERTA DO BRASIL
Celso Costa

SECRETÁRIO DE EDUCAÇÃO DO ESTADO DO PIAUÍ
Antônio José Medeiro

COORDENADOR GERAL DO CENTRO DE EDUCAÇÃO ABERTA A DISTÂNCIA DA UFPI
Gildásio Guedes Fernandes

SUPERITENDENTE DE EDUCAÇÃO SUPERIOR NO ESTADO
Eliane Mendonça

CENTRO DE CIENCIAS DA NATUREZA
Helder Nunes da Cunha

COORDENADOR DO CURSO DE SISTEMA DE INFORMAÇÃO NA MODALIADE DE EAD
Luiz Cláudio Demes da Mata Sousa

COORDENADORA DE MATERIAL DE DIDÁTICO DO CEAD/UFPI
Cleidinalva Maria Barbosa Oliveira

DIAGRAMAÇÃO
Joaquim Carvalho de Aguiar Neto

APRESENTAÇÃO

Um computador é uma máquina que manipula informações e o estudo de computação inclui a abordagem de como as informações são organizadas, manipuladas e utilizadas. Desta forma, um programa de computador consiste essencialmente de uma série de operações sobre um conjunto de dados. Esse conjunto de dados tem uma forma de estrutura que define como estes dados estão organizados internamente.

O objetivo desta apostila é proporcionar ao leitor um entendimento das Estruturas de Dados convencionais utilizadas nos programas de computador. O texto foi escrito de forma simples e objetiva. Cada capítulo é acompanhado de embasamento teórico e prático das estruturas, bem como de implementações e exercícios. A bibliografia e a webliografia ao fim das notas são mais do que suficiente para que o leitor se aprofunde na teoria apresentada em cada unidade.

Na **Unidade I** são apresentados os conceitos relacionados às estruturas de dados lineares, conhecidas como listas, pilhas e filas. Nela, descrevemos suas definições e principais operações relacionadas a estes tipos de estruturas.

Na **Unidade II** é descrita as estruturas adequadas para representarmos dados que devem ser dispostos de maneira hierárquica. Estas estruturas são conhecidas como árvores.

Por fim, na **Unidade III** é apresentado uma introdução à Teoria dos Grafos, enfocando os principais conceitos e aplicações.

SUMÁRIO GERAL

UNIDADE II – ÁRVORES

UNIDADE III – INTRODUÇÃO AOS GRAFOS

INTRODUÇÃO

Para resolver um problema utilizando o computador como instrumento é necessário que seja primeiramente encontrada uma maneira de descrever este problema de uma forma clara e precisa. É necessário que encontremos uma seqüência de passos que permitam que o problema possa ser resolvido de maneira automática e/ou repetitiva.

Além disto, é preciso definir como os dados a serem processados serão armazenados no computador. Desta forma, a solução de um problema por computador é baseada em dois pontos:

- a seqüência de passos e;
- a forma como os dados serão armazenados no computador.

Esta seqüência de passos é chamada de algoritmo. A palavra algoritmo se deriva da tradução ao latim da palavra árabe alkhowarizmi, nome de um matemático e astrônomo árabe que escreveu um tratado sobre manipulação de números e equações no

século IX. Sendo assim, podemos definir algoritmo como uma série de passos organizados (obedecendo a uma seqüência lógica) que descreve o processo que se deve seguir, para dar solução a um problema específico. Um exemplo simples de como um problema pode ser resolvido caso forneçamos uma seqüência de passos que mostrem a solução, é uma receita para preparar um bolo.

A noção de algoritmo é fundamental para a computação. O desenvolvimento de algoritmos para solucionar problemas é uma das maiores dificuldades dos iniciantes em programação em computadores. Este fato deve-se por não existir um conjunto de regras (um algoritmo) que nos permita criar algoritmos. Geralmente existem diversas formas para resolver o mesmo problema, cada um segundo o ponto de vista do seu criador.

Estruturas de Dados e algoritmos estão intimamente ligados. Não se pode estudar Estruturas de Dados sem considerar os algoritmos associados a elas, assim como a escolha dos algoritmos em geral depende da representação e da Estrutura de Dados. Sabe-se que algoritmos manipulam dados. Quando estes dados estão organizados de forma coerente, podemos caracterizá-los como uma Estrutura de Dados.

Estrutura de Dados é uma disciplina que trabalha com métodos de organização lógica dos dados, dando-lhes atributos mais funcionais do que se estivessem sem qualquer tipo de estruturação. Atualmente, uma grande fatia da performance dos grandes programas é dependente da eficiência das Estruturas de Dados utilizadas na sua implementação.

Os algoritmos irão manipular dados, que normalmente são fornecidos pelos usuários, e entregar resultados para eles. Uma pergunta importante neste momento é: que tipo de dados poderemos manipular?

As linguagens de programação definem seus tipos de dados, caracterizados como o conjunto de valores a que uma constante

pertence, ou que podem ser assumidos por uma variável ou expressão, ou que podem ser gerados por uma função.

Tipos simples de dados ou, simplesmente, Tipos de Dados, são grupos de valores indivisíveis, como os tipos básicos: inteiros, real, caracteres e lógicos. Por exemplo, uma variável do tipo inteiro pode assumir valores pertencentes ao conjunto dos números inteiros.

Geralmente as aplicações computacionais necessitam armazenar informações compostas de diversos Tipos de Dados, semelhantes ou não. Por exemplo, o cadastro de clientes de uma empresa, armazena informações do tipo: nome, endereço, telefone, identidade, etc. Para podermos armazenar esse tipo de informação, utilizamos um tipo conhecido como Tipo Abstrato de Dados (TAD).

Tipos Abstratos de Dados são tipos criados arbitrariamente pelo programador, sem que estejam definidos na linguagem de programação como Tipos de Dados primitivos.

Na realidade, os Tipos Abstratos de Dados são uniões de Tipos de Dados primitivos e, até mesmo, outros Tipos Abstratos de Dados, formando uma única estrutura, semelhante a uma ficha com vários campos.

Os Tipos Abstratos de Dados devem ser previamente declarados antes de serem utilizados, ou seja, antes de ser declarada uma variável com seu tipo.

Um TAD pode ser visto como um modelo matemático, acompanhado das operações definidas sobre o modelo. O conjunto dos inteiros acompanhado das operações adição, subtração e multiplicação forma um exemplo de um Tipo Abstrato de Dados.

Cabe ressaltar que cada conjunto diferente de operações define um Tipo Abstrato de Dados diferente, mesmo que todos os conjuntos de operações atuem sob um mesmo modelo matemático. Uma razão forte para isso é que a escolha adequada de uma

implementação depende fortemente das operações a serem realizadas sobre o modelo.

As Estruturas de Dados são chamadas Tipos de Dados compostos que se dividem em dois grupos: homogêneos (vetores e matrizes) e heterogêneos (registros). As estruturas homogêneas são conjuntos de dados formados pelo mesmo Tipo de Dado primitivo. As estruturas heterogêneas são conjuntos de dados formados por Tipos de Dados primitivos diferentes em uma mesma estrutura. A escolha de uma Estrutura de Dados apropriada pode tornar um problema complicado em um de solução bastante trivial. O estudo das Estruturas de Dados está em constante desenvolvimento mas, apesar disso, existem certas estruturas clássicas que se comportam como padrões.

Esta apostila proporcionará ao leitor um conhecimento das Estruturas de Dados clássicas utilizadas nos programas de computador. Ao longo dos capítulos iremos abordar as Estruturas de Dados convencionais conhecidas como: Listas Lineares, Árvores e Grafos.

Boa Leitura!!

André Macedo Santana

Erico Meneses Leão

Esta unidade é dedicada às estruturas de dados lineares conhecidas como Listas, Filas e Pilhas. Uma Lista é uma estrutura de dados caracterizada por conter dados ou informações dispostos um após o outro, em uma ordem estritamente linear. Já as Pilhas e Filas são estruturas de dados que têm semelhanças de implementação, diferenciando-se na prioridade de remoção de um de seus elementos.

Enquanto que uma Pilha implementa uma política do último a chegar é o primeiro a sair (*last in first out - LIFO*), a Fila implementa a política do primeiro a entrar é o primeiro a sair (*first in first out - FIFO*).

Nesta unidade apresentaremos as definições para Listas, Filas e Pilhas, abordando seus principais tipos, além de suas características, vantagens, desvantagens, operações, implementações, exemplos e aplicações práticas do nosso dia-a-dia, a fim de disponibilizar o embasamento teórico suficiente para o leitor.

Cada capítulo é acompanhado de definições, práticas, desafios e diversos exercícios, a fim de proporcionar conhecimento teórico e prático ao leitor. A bibliografia e a webliografia ao fim das notas são mais do que suficiente para adquirir um conhecimento razoável de listas lineares.

SUMÁRIO

UNIDADE I – ESTRUTURA DE DADOS LINEARES

UNIDADE I

ESTRUTURAS DE DADOS LINEARES

1. LISTAS LINEARES

1.1. Introdução

Uma lista é uma estrutura que armazena elementos que são acessíveis um após o outro, em ordem estritamente linear. São estruturas formadas por um conjunto de dados de forma a preservar a relação de ordem linear entre eles. Como exemplo de uma lista, podemos destacar uma lista de empregados de uma determinada empresa, uma lista de peças de uma revendedora, uma lista de compras de supermercado, etc.

Uma lista é composta por elementos, comumente chamados de nós, os quais podem conter cada um deles, um dado primitivo ou composto, como mostra a figura 2.1.

Figura 2.1: Representação de um nó de uma lista.

DEFINIÇÃO: Uma lista linear é um conjunto de n nós (n >=0) x_1, x_2, x_3,..., x_n, organizados estruturalmente de forma a refletir as posições relativas dos mesmos: Se n>0 então x_1 é o primeiro nó; para 1<k<n, tem-se que o nó x_k é precedido pelo nó x_{k-1} e é sucedido pelo nó x_{k+1}; e x_n é o último nó. Caso n=0, a lista é considerada vazia. A figura 2.2 esboça esta configuração.

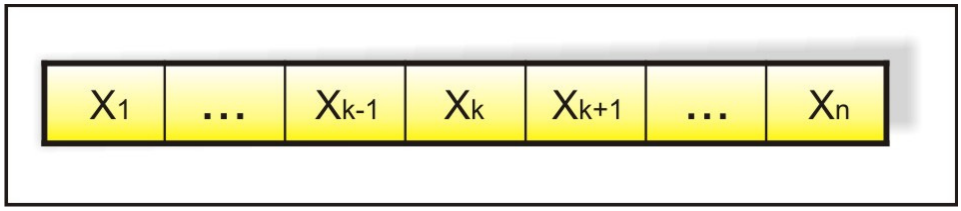


Figura 2.2: Representação de uma lista linear.

Operações em listas lineares

Uma vez definido que um conjunto de dados será representado sob a forma de uma lista linear, precisamos decidir as operações possíveis de serem implementadas sobre esta lista de dados. As operações mais comuns são definidas a seguir:

- Criar uma lista vazia.
- Verificar se uma lista está vazia.
- Determinar o número de nós de uma lista.
- Inserir um novo nó na lista (antes ou depois do k-ésimo nó).
- Acessar o k-ésimo nó da lista.
- Obter a posição de um nó da lista cujo valor é fornecido.
- Remover um nó da lista.
- Dividir uma lista em dois ou mais listas.
- Concatenar duas listas.
- Exibir todos os nós de uma lista.

Formas de representação

Existem várias formas possíveis de se representar internamente uma lista linear. A escolha de uma dessas formas dependerá da freqüência com que determinadas operações serão executadas sobre a lista, uma vez que algumas representações são favoráveis a algumas operações, enquanto que outras não o são, no sentido de exigir maior esforço computacional para sua execução.

Dessa forma, uma lista pode ser representada, basicamente, de duas formas:

- Seqüencial ou por Contigüidade dos nós:

Uma lista com representação seqüencial ou por contigüidade é caracterizada por seus nós serem implementados de forma consecutiva na memória física. Assim, este tipo de lista explora a seqüencialidade da memória do computador, de tal forma que seus nós sejam armazenados em endereços seqüenciais, ou igualmente distanciados um do outro. Pode ser representada por um vetor (figura 2.3) na memória principal ou um arquivo seqüencial em memória secundária.

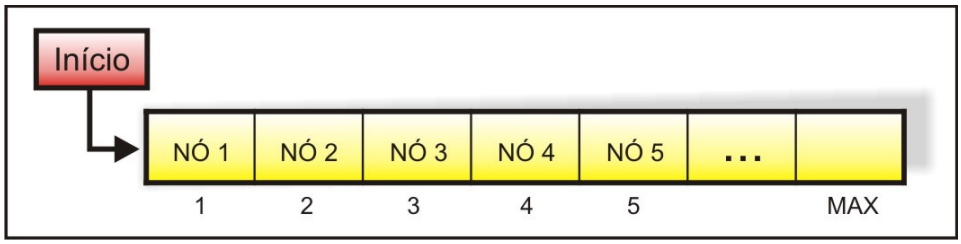


Figura 2.3: Representação de uma lista por contigüidade dos nós.

- Por encadeamento dos nós:

Uma lista com representação por encadeamento é caracterizada por seus nós serem implementados em qualquer posição na memória física, porém, cada nó, além de possuir a sua informação específica, contém, também, um ponteiro (endereço na memória) do próximo nó da seqüência, como mostra a figura 2.4.

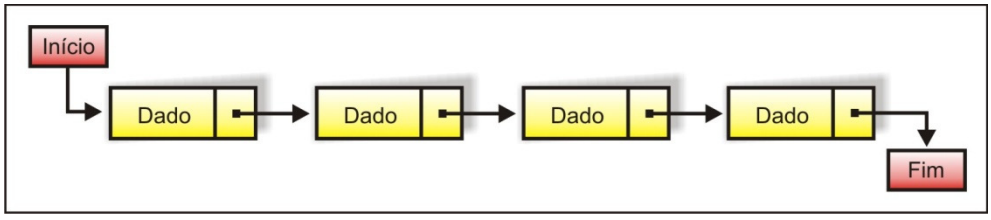


Figura 2.4: Representação de uma lista por encadeamento dos nós.

1.2. Lista Seqüencial

Uma lista com representação seqüencial é um conjunto de registros (nós) onde estão estabelecidas regras de precedência, de tal forma que esses nós sejam armazenados em endereços contíguos ou igualmente distanciados um do outro.

Características

A relação de ordem entre os nós da lista é representada pelo fato de que se o endereço do nó x_i é conhecido, então o endereço do nó x_{i+1} pode ser determinado.

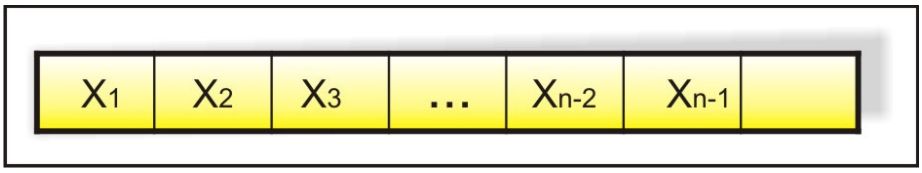


Figura 2.5: Lista seqüencial.

Como observado na figura 2.5, se o endereço do nó xi é conhecido e tomado como referência, podemos chegar também a todos os outros nós da lista.

Dessa forma, as seguintes características podem ser destacadas:

- Os nós na lista estão armazenados na memória fisicamente em posições consecutivas.
- A inserção de um novo nó na posição i causa o deslocamento à direita do nó x_i ao último.
- A remoção de um nó da posição i causa o deslocamento à esquerda do nó x_{i+1} ao último.

Uma lista seqüencial pode ser considerada vazia ou pode ser escrita como uma seqüência de nós pertencente a um mesmo conjunto.

Vantagens e desvantagens de usar listas seqüenciais

Listas seqüenciais é um tipo abstrato de dados adequado quando se trata de listas pequenas, com tamanhos bem definidos e inserções/remoções no final da lista.

A principal vantagem de se utilizar listas seqüenciais é que podemos ter acesso direto indexado a qualquer nó da lista. Porém, devido às suas características estáticas, uma lista seqüencial deverá possuir um tamanho máximo pré-definido, além de exigir várias movimentações nas operações de inclusão/remoção aleatórias, o que passa a ser desvantajoso quando tratamos de listas com tamanhos elevados.

Definição da estrutura de dados de listas seqüenciais

Uma lista linear seqüencial é definida como um vetor, no qual "*tipo"* é o tipo de dado a ser representado em cada nó da lista (tipos de dados dependente da linguagem de programação).

Dessa forma, temos:

```
Constante
        m = 100 //número de nós de uma lista
Tipo
        Lista = vetor [1..m] de tipo;
```

Operações em listas seqüenciais

A partir da definição da estrutura de dados de uma lista seqüencial, este tópico aborda algumas operações comuns sobre listas seqüenciais e os algoritmos para implementá-las.

a) acessar o *k-ésimo* nó de uma lista: através deste procedimento é possível acessar qualquer nó de uma lista seqüencial.

```
Procedimento acessar(L: Lista; k,N: inteiro; ref dado: tipo)
// L é o tipo Lista, k será utilizado para índice, N é referente à quantidade
//de elementos no vetor e dado é a variável que guarda a informação da
lista
Início
        //teste para evitar escolher uma posição que não pertença ao vetor
        se (k <= 0) ou (k > N) então
                imprima("Dimensão invalida");
        senão
                dado = L[k];
fim
```

b) alterar o valor do *k-ésimo* nó de uma lista: através deste procedimento é possível alterar o valor (dado) de qualquer elemento da lista seqüencial.

```
Procedimento alterar(L: Lista; k,N: inteiro; ref dado: tipo)
Início
        //teste para evitar escolher uma posição que não pertença ao vetor
        se (k <= 0) ou (k > N) então
                imprima("Dimensão invalida");
        senão
                L[k] = dado;
fim
```

c) inserir um novo nó em uma lista: a inserção de um novo nó em uma lista seqüencial pode ocorrer antes do primeiro elemento ou antes do *k-ésimo* elemento. O procedimento a seguir mostra a inserção de um novo elemento antes do k-ésimo elemento fornecido.

```
Procedimento inserir(L: Lista; k,N: inteiro; ref dado: tipo)
Início
        variável i : inteiro;
        //teste para evitar escolher uma posição que não pertença ao vetor
        se (k <= 0) ou (k > N) então
                imprima("Dimensão invalida");
        senão
                para i de fim até k passo -1 faça
                        L[i+1] = L[i];
                N = N + 1;      // define um elemento a mais no vetor
                L[k] = dado;
fim
```

d) remover o *k-ésimo* nó de uma lista: este procedimento é responsável por remover o nó *k* especificado da lista seqüencial.

```
Procedimento remover(L: Lista; k,N: inteiro; ref dado: tipo)
Início
        variável i : inteiro;
        //teste para evitar escolher uma posição que não pertença ao vetor
        se (k <= 0) ou (k > N) então
                imprima("Dimensão invalida");
        senão
                para i de k até fim passo -1 faça
                        L[i] = L[i+1];
                N = N - 1;      // define um elemento a menos no vetor
fim
```

Praticar: explique a finalidade da estrutura de repetição (para) nos procedimentos de inserção e remoção de nós na lista seqüencial.

DESAFIO: elabore um procedimento de inserção no inicio de uma lista seqüencial.

1.3. Lista Encadeada

Vimos no tópico 1.2 que é possível representar listas através de estruturas estáticas (vetor). O vetor é a forma mais primitiva de representar diversos elementos agrupados. Porém, vimos também que a utilização de vetores para representar listas apresenta uma série de desvantagens, já comentadas anteriormente, devido suas características estáticas.

Uma forma de permitir o crescimento dinâmico do comprimento máximo da lista, ou seja, à medida que for necessário mais espaço físico de memória para crescimento, este espaço vai sendo alocado em tempo de execução, bem como diminuir o esforço computacional das operações de inserção e remoção, é representar a lista por ENCADEAMENTO. Na lista por encadeamento, os nós são ligados entre si para indicar a relação existente entre eles (relação de ordem). Cada nó da lista deverá conter além das

informações respectivas, uma indicação ao nó seguinte. Essas listas são chamadas de listas encadeadas (ligadas), também conhecidas como lista simplesmente encadeada.

Características

Para manipular uma estrutura sem ter de alterar todos seus vizinhos, precisamos de algo que não seja fixo. Um vetor é uma estrutura fixa na memória, precisamos sempre empurrar todos seus vizinhos para um lado para conseguir espaço, ou para remover algum dos elementos, para não deixar espaços vazios.

O elemento básico de uma lista encadeada é o nó. Cada nó da lista é formado por uma parte que armazena dados e outra que armazena campo de ligação. O campo de ligação de uma lista encadeada possui o endereço de memória física onde o próximo nó está armazenado, permitindo assim o encadeamento (ordem de acesso) dos dados e possibilitando a implementação de listas encadeadas. A figura 2.6 ilustra um nó de uma lista encadeada.

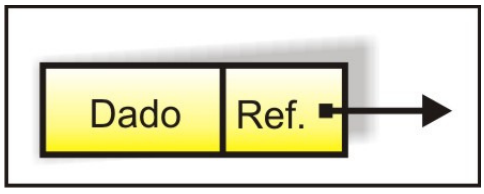


Figura 2.6: Nó de uma lista encadeada.

Numa lista encadeada, para cada novo nó inserido, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela lista é proporcional ao número de nós nela armazenado. A figura 2.7 mostra uma lista encadeada com 5 nós alocados.

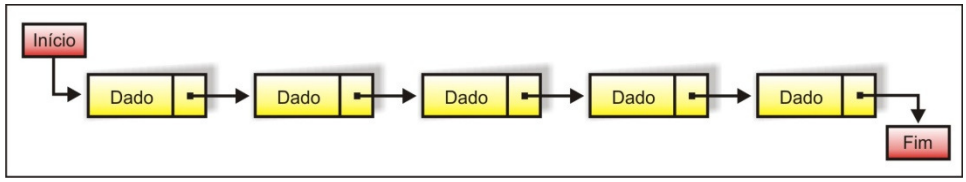


Figura 2.7: Lista encadeada com 5 nós alocados.

No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos desta lista, devemos explicitamente guardar o encadeamento dos nós (ordenamento), o que é feito armazenando-se, junto com a informação de cada elemento, uma referência para o próximo elemento da lista. A figura 2.8 mostra esta estrutura.

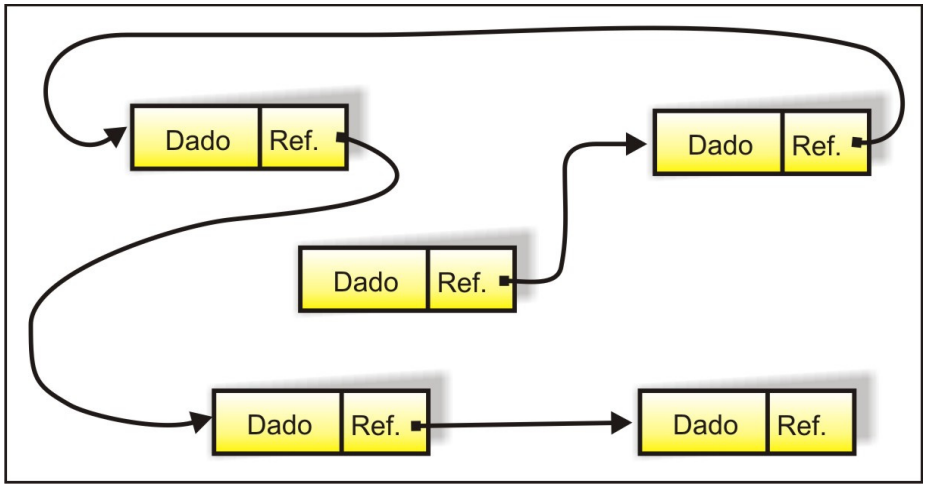


Figura 2.8: Representação de uma Lista encadeada.

Vantagens e desvantagens de usar listas encadeadas

A principal vantagem de se utilizar listas encadeadas está no fato de não ser preciso reservar uma área de memória com tamanho fixo. Além disso, nas operações de inserção/remoção não é necessário deslocamentos na lista.

Entretanto, um nó da lista encadeada ocupa mais espaço em memória do que um elemento (nó) de um vetor, devido à necessidade de guardar informações a mais (referência para o próximo nó da lista, por exemplo). Embora este espaço a mais não ocupar um espaço muito grande, existe este gasto a mais de memória. Além disso, o acesso a um nó da lista encadeada geralmente é mais demorado que em um vetor. Assim, o acesso a um nó de um vetor é direto ou indexado, enquanto que para acessar um nó de uma lista é necessário percorrer todos os nós da lista até chegar ao nó desejado.

Definição da estrutura de dados de listas encadeadas

Supondo que cada nó da lista encadeada contenha uma variável *dado* (que guarda a informação desta lista, o qual seu tipo é dependente da linguagem de programação utilizada) e uma variável *próximo*, variável utilizada para fazer referência ao próximo nó da seqüência, podemos definir um nó desta lista como a seguir.

Tipo
nó::reg(dado: tipo; próximo:ref nó);

Desta forma, cada novo nó da lista será um agregado do tipo registro, contendo dois campos: um para a informação (*dado*) e o outro para a referência ao próximo nó do encadeamento (*próximo*), como mostra a figura 2.9.

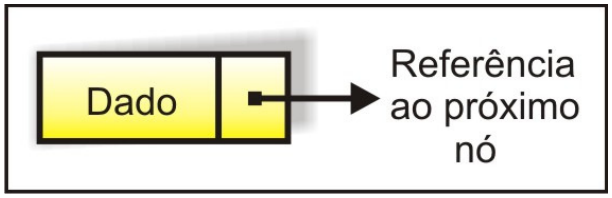


Figura 2.9: Nó de uma lista encadeada com a referência para próximo nó.

Em JAVA, podemos definir a classe nó como sendo:

```
class No
{
public int dado; //um nó com dado inteiro
public No próximo; //referência para o próximo nó
}
```

Para que seja possível implementar esta estrutura, devemos ter uma variável do tipo referência a nó, para indicar o início da lista encadeada.

```
variável início:ref nó;

private No inicio; //em JAVA
```

Antes de iniciar o procedimento de inserção de novos elementos na lista, a variável *início* precisa ser iniciada com valor vazio (NIL), indicando que a lista está fazia.

```
início = NIL;

inicio = null; //em JAVA
```

Note que a variável *início* deve sempre apontar para o início da lista, informando onde a lista encadeada começa, não sendo necessário armazenar nenhum valor de informação, como mostra a figura 2.10.

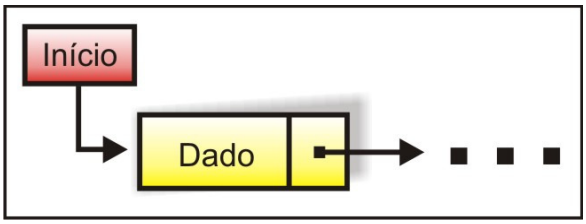


Figura 2.10: Referência para o início de uma lista encadeada.

Quando se deseja inserir um novo nó na lista, primeiramente, é preciso alocar um espaço de memória para o referido nó, que chamaremos de *nó p*:

var p: ref nó;
alocar(p);

e, em seguida, definir os valores da informação e da referência deste novo nó p. A seguir são definidas as estruturas de atribuição de valores para o nó:

p↑.dado
p↑.próximo

Operações em listas encadeadas

Há duas alternativas para implementação de operações de listas encadeadas: utilizando vetores estáticos ou utilizando alocação dinâmica. Entretanto, não há vantagem na implementação da lista encadeada com vetores, uma vez que permanece o problema do tamanho máximo da lista predefinido além da necessidade de utilizar mais espaço para armazenar os endereços dos elementos sucessores. Portanto, vamos utilizar implementação dinâmica para as nossas listas encadeadas.

a) criação de uma lista simplesmente encadeada vazia: o procedimento a seguir mostra a definição e criação de uma lista simplesmente encadeada, inicialmente, sem nenhum nó alocado.

```
Tipo
        nó::reg(dado: tipo, próximo:ref nó),
Início
        variável início: ref nó;
        início = NIL;
fim
```

b) inserção de um elemento na lista simplesmente encadeada: os passos a seguir são suficientes para inserir um novo elemento sempre no início da lista simplesmente encadeada.

(1) *variável p: ref nó;*

(2) *alocar(p);*

(3) *p↑.próximo = início;*

(4) *início = p;*

(5) *p↑.dado = valor; //valor a ser armazenado neste nó*

Considerando que já exista uma lista simplesmente encadeada com apenas dois nós alocados, estes passos produzem o desencadeamento de uma série de etapas descrita a seguir:

(1) nesta linha é definido um novo nó chamado de *p*.

(2) neste momento, o nó *p* é alocado e reservado um espaço de memória física suficiente para guardar todas as suas informações, como mostra a figura 2.11.

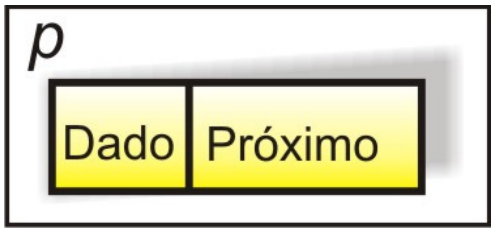


Figura 2.11: Alocando um novo p para ser inserido na lista encadeada.

(3) esta linha, faz com que a referência *próximo* do novo nó aponte para onde a variável *início* estava referenciando, como mostra a figura 2.12.

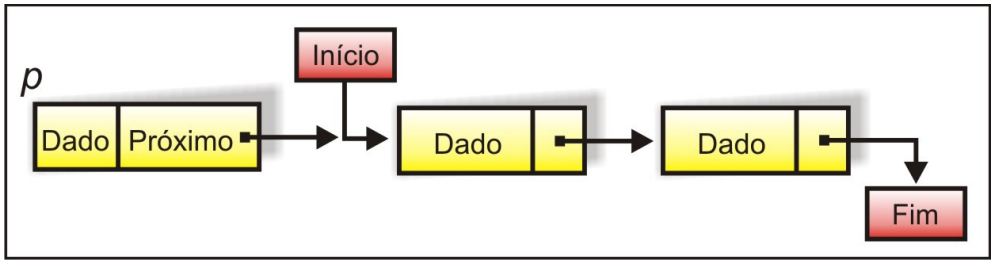


Figura 2.12: Processo de inserção de um novo nó na lista encadeada.

(4) por fim, a variável *início* passa a apontar para o novo nó alocado, indicando o novo início da lista, como visualizado na figura 2.13.

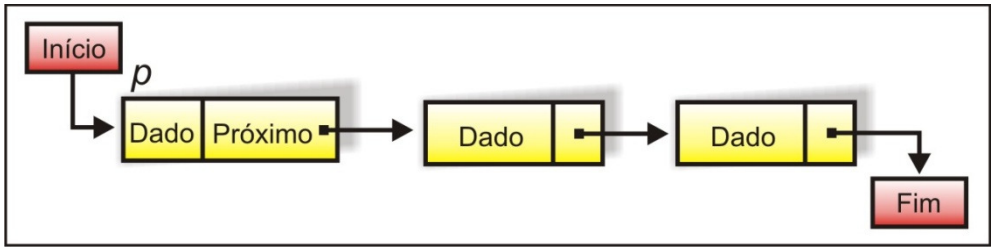


Figura 2.13: Lista encadeada depois do processo de inserção de um novo nó no início.

(5) esta instrução serve para atribuir um valor para o novo nó inserido nesta lista.

Para o procedimento de inserção no início, temos o seguinte método em JAVA:

```
public void inserir_inicio(int dado)
{
No novoNo = new No(dado);
novoNo.proximo = inicio; //novo nó aponta para o inicio antigo
inicio = novoNo; //o inicio aponta agora para o novo nó
}
```

c) remoção do primeiro elemento na lista simplesmente encadeada: o procedimento de remoção do primeiro nó da lista é utilizado para remover um nó à esquerda na lista, liberando a área de memória reservada para este nó e atualizando o conteúdo da variável *início* com a referência do próximo nó da lista. Os passos a seguir referem-se à remoção de do primeiro elemento de uma lista simplesmente encadeada.

(1) *início = p↑.próximo;*
(2) *desalocar(p);*

Note que esses passos são suficientes para a remoção do primeiro elemento de uma lista simplesmente encadeada. No passo (1) a variável *início* passa a referenciar o nó para o qual o primeiro

elemento apontava (ou seja, o segundo nó). Após esse passo, basta desalocar o espaço antes ocupado pela variável *p*. A figura 2.14 ilustra esse procedimento.

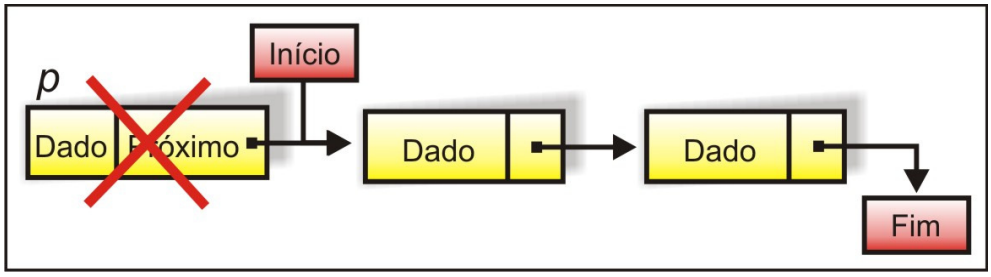


Figura 2.14: A variável *início* apontando para o segundo elemento da lista e o primeiro elemento da lista sendo desalocado.

Para o procedimento de inserção no início, temos o seguinte método em JAVA:

```
public No remover_inicio()
{
No aux = inicio;
inicio = inicio.proximo; //novo inicio definido para o segundo elemento
return aux; //retorna o nó removido
}
```

Praticar: elabore procedimentos de edição e procura de nós na lista e outro procedimento que informe a quantidade de elementos existentes na lista simplesmente encadeada.

DESAFIO: elabore um procedimento de remoção do último nó de uma lista simplesmente encadeada. Para facilitar o desenvolvimento, crie uma nova variável "fim" do tipo ref nó, que aponte sempre para o último elemento da lista.

1.4. Lista Duplamente Encadeada

A estrutura de lista simplesmente encadeada vista na seção anterior caracteriza-se por formar um encadeamento simples entre os seus nós. Dessa forma, cada nó desta lista contém uma referência para o nó seguinte, o que nos permite percorrer toda a lista encadeada a partir do primeiro nó (do início para o fim). Porém, neste tipo de lista, não temos uma eficiência para percorrer a lista do fim para o início. Ainda mais, a remoção de um elemento de uma lista simplesmente encadeada também é uma tarefa difícil. Mesmo conhecendo a posição do elemento a ser removido, teremos que percorrer toda a lista, nó por nó, a fim de encontrarmos o elemento anterior, pois, dado um determinado nó, não temos como acessar diretamente o nó anterior.

A fim de solucionar esses problemas, podemos utilizar uma estrutura de dados conhecida como *lista duplamente encadeadas*. Uma lista duplamente encadeada é aquela em que cada nó possui duas Referências, ao invés de uma só. A primeira é usada para indicar o nó sucessor (da mesma forma que lista simplesmente encadeada), enquanto que a segunda é usada para apontar para o nó predecessor (nó anterior). Desta forma, dado um nó qualquer da lista, podemos acessar ambos os nós adjacentes: o próximo e o anterior. A figura 2.15 ilustra este tipo de lista.

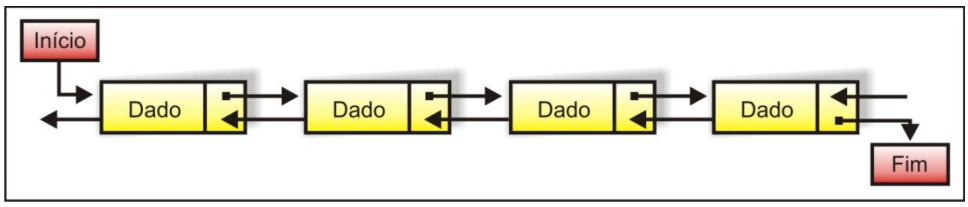


Figura 2.15: Lista duplamente encadeada.

Observe através da figura 2.15, que se tivermos uma referência para o último elemento da lista, podemos percorrer toda a

lista em ordem inversa, bastando acessar continuamente o nó anterior, até alcançar o primeiro nó da lista.

Vantagens e desvantagens de usar listas duplamente encadeadas

A principal vantagem de se utilizar listas duplamente encadeadas é que, devido à existência da referência para o nó anterior e o posterior, podemos percorrer facilmente toda a lista em qualquer direção (do início para o fim e do fim para o início). Com isso, facilita a implementação da maioria das operações possíveis em cima de listas.

Entretanto, uma lista duplamente encadeada ocupa ainda mais espaço em memória, pois cada nó conterá uma referência a mais (para o nó anterior).

Definição da estrutura de dados de listas duplamente encadeadas

Supondo que cada nó da lista duplamente encadeada deva conter um dado de um tipo qualquer (tipos disponíveis dependendo da linguagem de programação), podemos definir um nó desta lista como a seguir:

Novo_tipo nó::reg(dado: tipo;

anterior:ref nó;

próximo:ref nó);

Onde dado é a informação definida pelo usuário, anterior é a referência para o nó anterior e próximo é a referência para o próximo nó da lista.

Desta forma, cada novo nó da lista será um agregado do tipo registro, contendo três campos: um para a informação e os outros

dois para referências ao nó anterior e próximo do encadeamento, como mostra a figura 2.16.

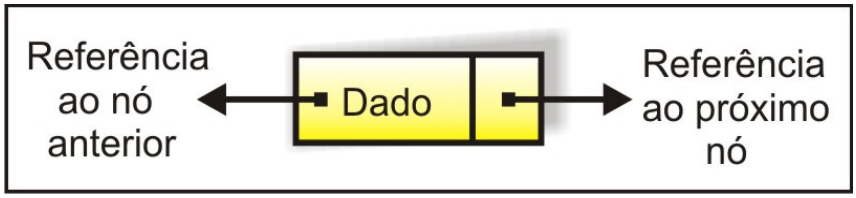


Figura 2.16: Um nó de uma lista duplamente encadeada com referência para o nó próximo e anterior.

Em JAVA, podemos definir a classe nó como sendo:

```
class No
{
public int dado; //um nó com dado inteiro
public No próximo; //referência para o próximo nó
public No anterior; //referência para o nó anterior
}
```

Para que seja possível implementar esta estrutura, devemos também ter uma variável do tipo referência a nó, para indicar o início da lista encadeada:

```
variável início:ref nó;

private No inicio; //em JAVA
```

Operações em listas duplamente encadeadas

As operações em listas duplamente encadeadas são análogas às operações de listas simplesmente encadeadas. A principal diferença refere-se ao fato de que cada nó de uma lista duplamente encadeada possuir, além de uma referência ao nó posterior, uma referência ao nó anterior. Esta referência a mais facilita na definição de uma série de operações sobre este tipo de

lista, principalmente as operações de busca de nós na lista, pois permite, a partir de qualquer nó, o retorno a nós anteriores.

a) criação de uma lista duplamente encadeada vazia: como em listas simplesmente encadeada, para se criar uma lista encadeada é preciso apenas definir que a variável *início* do tipo *ref nó* referencie vazio (*NIL*), criando, assim, uma lista duplamente encadeada sem nenhum nó alocado.

```
Tipo
        nó::reg(dado: tipo; próximo:ref nó; anterior:ref nó),
Início
        variável início: ref nó;
        início = NIL;
fim
```

b) inserção de um elemento na lista duplamente encadeada: os passos a seguir são suficientes para inserir um novo elemento sempre no início da lista duplamente encadeada.

```
(1) variável p: ref nó;
(2) alocar(p);
(3) p↑.dado = valor; //valor a ser armazenado neste nó
(4) p↑.próximo = início;
(5) p↑.anterior = NIL;
(6) início↑.anterior = p;
(7) início = p;
```

Considerando que já exista uma lista simplesmente encadeada com dois nós alocados, estes passos produzem o desencadeamento de uma série de etapas descrita a seguir:

(1) nesta linha é definido um novo nó chamado de *p*.

(2) neste momento, o nó p é alocado e reservado um espaço de memória física suficiente para guardar todas as suas informações, como mostra a figura 2.17.

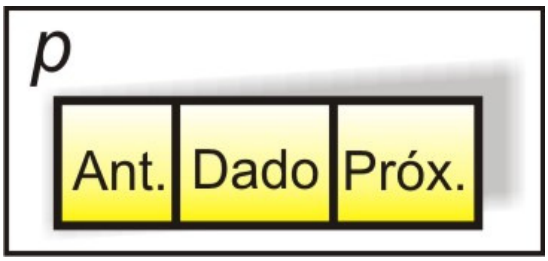


Figura 2.17: Alocando um novo *p* para ser inserido na lista duplamente encadeada.

(3) esta instrução serve para atribuir um valor para o novo nó inserido nesta lista.

(4) esta linha faz com que a referência *próximo* do novo nó aponte para onde a variável $início$ estava referenciando, como mostra a figura 2.18.

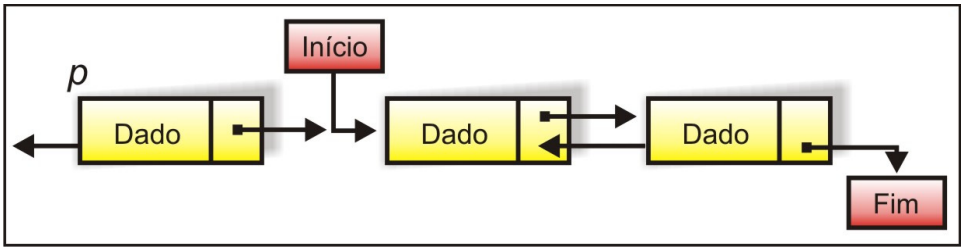


Figura 2.18: Processo de inserção de um novo nó na lista encadeada.

(5) esta linha faz com que a referência *anterior* do novo nó aponte para vazio (figura 2.18).

(6) este passo faz com que a referência *anterior* de início (primeiro nó da lista duplamente encadeada) aponte para o novo nó *p* que se deseja inserir na lista, como pode ser visualizado na figura 2.19.

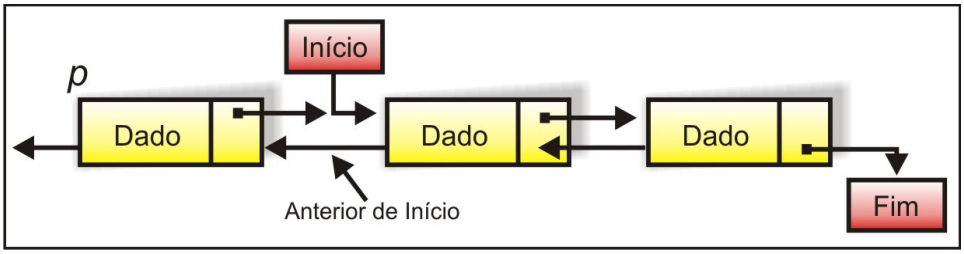


Figura 2.19: Ajustando a referência anterior do nó que a variável início referencia.

(7) por fim, a variável *início* passa agora a apontar para o novo nó alocado, indicando o novo início da lista, como visualizado na figura 2.20.

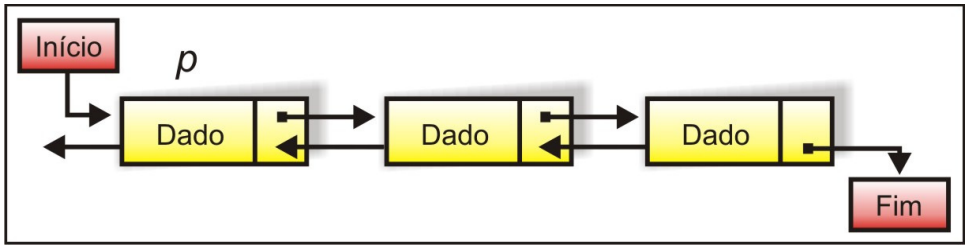


Figura 2.20: Lista encadeada depois do processo de inserção de um novo nó no início.

Para o procedimento de inserção no início, temos o seguinte método em JAVA:

```
public void inserir_inicio(int dado)
{
No novoNo = new No(dado);
inicio.anterior = novoNo; //faz com que o a referência anterior do inicio
                          //antigo aponte para o novo nó
novoNo.proximo = inicio; //novo nó aponta para o inicio antigo
inicio = novoNo; //o inicio aponta agora para o novo nó
}
```

Praticar: elabore procedimentos de edição e procura de nós em uma lista duplamente encadeada.

DESAFIO: elabore um procedimento de remoção do primeiro nó de uma lista duplamente encadeada.

1.5. Lista Encadeada Circular

Algumas aplicações necessitam representar conjuntos cíclicos. Por exemplo, as arestas que delimitam uma face podem ser agrupadas por uma estrutura circular. Para esses casos, podemos usar *listas encadeadas circulares*.

Uma lista circular assemelha-se muito com uma lista encadeada, diferenciando-se pelo fato de seu último elemento conter uma referência para o primeiro elemento, formando a situação circular, como pode ser observado na figura 2.21.

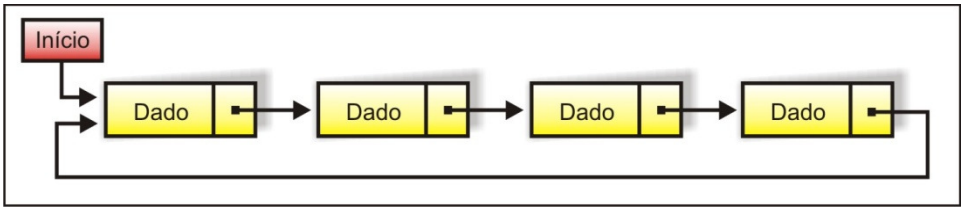


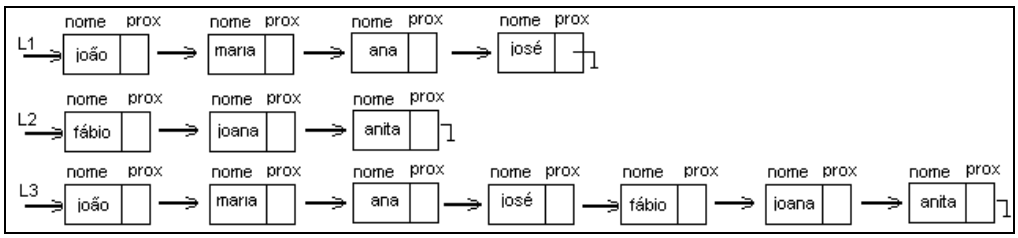
Figura 2.21: Lista circular.

Para percorrer os nós de uma lista circular, partimos da referência do nó inicial até alcançarmos novamente esse mesmo nó.

DESAFIO: baseado em uma lista simplesmente encadeada circular, elabore um procedimento de inserção e remoção do primeiro nó da lista e um procedimento de percorrer a lista completa a quantidade de vezes definidas pelo usuário.

1.6. Exercícios

1. Defina lista linear.
2. Qual a principal diferença entre lista por contigüidade e lista por encadeamento dos nós?
3. Descreva com detalhes as vantagens e desvantagens de se utilizar lista por contigüidade.
4. Descreva com detalhes as vantagens e desvantagens de se utilizar lista por encadeamento.
5. Qual a principal vantagem em se utilizar a estrutura de lista duplamente encadeada?
6. Faça uma procedimento que conte quantos elementos existem em uma lista simplesmente encadeada.
7. Escreva um programa que crie duas listas (L1 e L2). Em seguida, o seu programa deve calcular a união das duas listas, ou seja, construir uma lista L3, contendo todas as informações presentes em L1 e todas as informações presentes em L2. As listas L1 e L2 devem permanecer inalteradas. Veja o exemplo abaixo.



8. Faça um programa que contenha as principais operações do Tipo Abstrato de Dados Lista Simplesmente Encadeada. Esta lista deverá ser de números inteiros (tipo do dado da lista). O seu programa deverá conter as seguintes operações:
    - criar lista simplesmente encadeada;
    - verificar se a lista está vazia;
    - adicionar um novo nó na lista (no início da lista);

- remover um nó qualquer da lista;
- verificar se existe um nó qualquer na lista.

9. Faça um programa que contenha as principais operações do Tipo Abstrato de Dados Lista Duplamente Encadeada. Esta lista deverá ser de números inteiros (tipo do dado da lista). O seu programa deverá conter as seguintes operações:
    - criar lista duplamente encadeada;
    - verificar se a lista está vazia;
    - adicionar um novo nó na lista (no início da lista);
    - remover um nó qualquer da lista;
    - verificar se existe um nó qualquer na lista;
    - Procurar um nó qualquer na lista e imprimir seu sucessor e antecessor (caso existam).
10. Faça um programa que contenha as principais operações da Estrutura de Dados do tipo Lista Circular. Esta lista deverá ser de números inteiros (tipo do dado da lista). O seu programa deverá conter as seguintes operações:
    - criar lista simplesmente encadeada circular;
    - verificar se a lista está vazia;
    - adicionar um novo nó na lista (no início da lista);
    - remover um nó qualquer da lista;
    - verificar se existe um nó qualquer na lista;
    - Imprimir da lista (quantidade de vezes exigida pelo usuário)
11. (QUESTÃO DESAFIO) Suponha que uma empresa de material de construção necessite desenvolver um software para gerenciamento de seu estoque. Cada produto que entra na empresa recebe um código (número inteiro único), além da

descrição do produto (string) e a quantidade presente no estoque. Para o armazenamento das informações é necessário a definição de estruturas de dados eficazes e dinâmicas, a fim de otimizar o gasto de memória. Construa um esboço do software utilizando a Estrutura de Dados do tipo Lista simplesmente encadeada, com as seguintes operações:

- Criação da lista;
- Verificar se existe ou não algum material cadastrado (verificar se a lista é vazia);
- Inserir um novo item (no fim da lista);
- Impressão da lista completa.

12. Para a questão anterior, construa um esboço do software utilizando a Estrutura de Dados do tipo Lista duplamente encadeada, com as seguintes operações:
    - Criação da lista;
    - Inserir um novo item (no início da lista);
    - Buscar um item através de seu código;
    - Impressão da lista completa.
13. Para a mesma questão (Questão 13), construa um esboço do software utilizando a Estrutura de Dados do tipo Lista circular (simplesmente encadeada) com as seguintes operações:
    - Criação da lista;
    - Inserir um novo item (no fim ou início da lista);
    - Impressão da lista (quantidade de vezes exigida pelo usuário).

UNIDADE I

ESTRUTURAS DE DADOS LINEARES

2. PILHAS

2.1. Introdução

A pilha é uma das estruturas de dados mais simples. Por esse motivo, ela é bastante utilizada em programação, podendo ser implementada diretamente pelo *hardware* da maioria das máquinas modernas.

As pilhas é um tipo de estrutura de dados linear que têm sua utilização e implementação semelhantes às listas, sendo que diferem apenas na prioridade da retirada da informação.

A idéia básica de uma pilha é que todo o acesso aos seus nós são realizadas através do seu topo. Essa política de acesso é conhecido LIFO ("*last in first out*" – "último a chegar primeiro a sair"). Nesta política, quando um novo nó é introduzido na pilha, passa a ser o elemento do topo, e o único nó que pode ser removido da pilha é o do topo. Isto faz com que os nós da pilha sejam retirados na ordem inversa à ordem em que foram introduzidos.

Para facilitar o entendimento de pilhas, podemos imaginar uma pilha de livros, como mostra a figura 2.22.

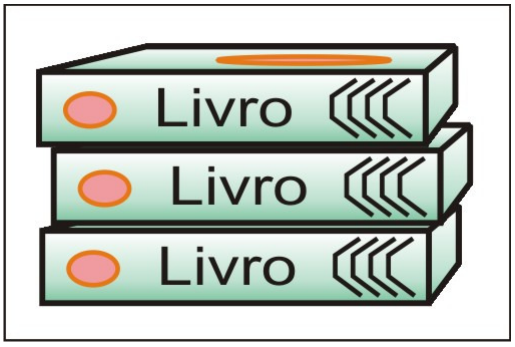


Figura 2.22: Pilha de livros.

Para adicionar um novo livro na pilha, o colocamos no topo. Da mesma forma, para retirarmos um livro da pilha, retiramos do topo. A figura 2.23 mostra a operação de inserção de dois novos livros (a) e retirada (b) de um desses livros da pilha.

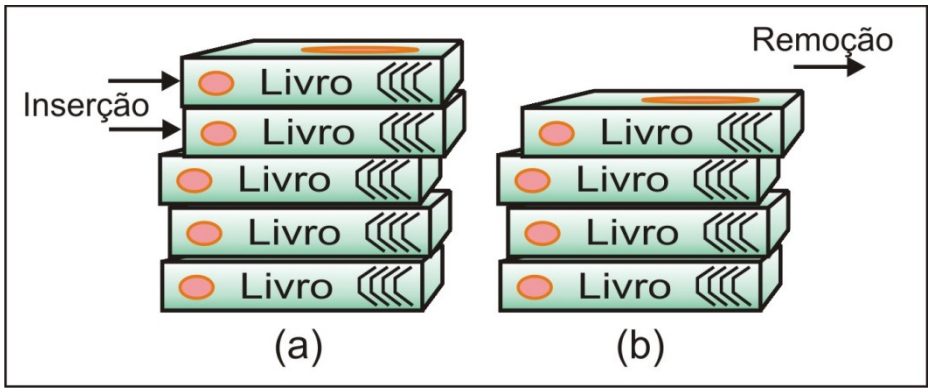


Figura 2.23: (a) Procedimento de inserção de dois livros na pilha e (b) procedimento de remoção de um livro da pilha.

Existem duas operações básicas que devem ser implementadas numa estrutura de dados pilha:

- empilhar um novo elemento: inserção do elemento no topo;
- desempilhar um elemento: remoção do elemento do topo.

Comumente, são utilizados os termos em inglês *push* (empilhar) e *pop* (desempilhar). A figura 2.24 mostra as operações básicas de uma pilha.

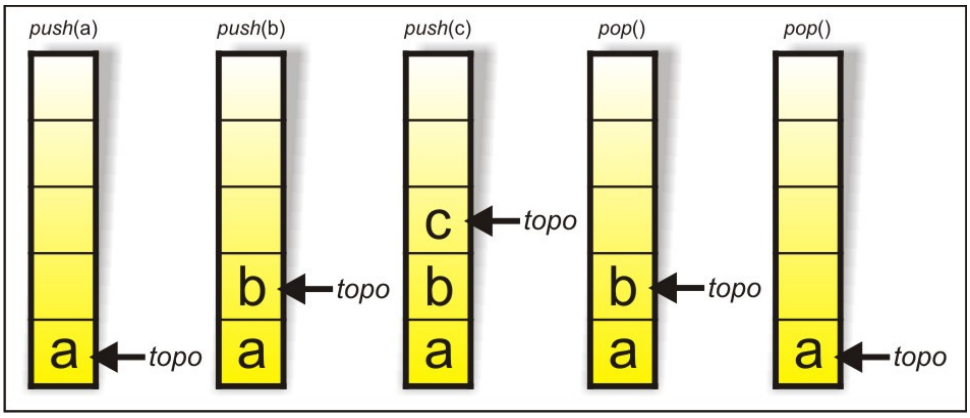


Figura 2.24: Operações básicas de uma pilha (*push* e *pop*).

DEFINIÇÃO: dada um pilha P = (a_1, a_2, ..., a_n), dizemos que a_1 é a base da pilha; a_n é o elemento topo da pilha; e a_{i+1} está acima de a_i.

Operações associadas a pilhas

Uma vez definido que um conjunto de dados será representado sob a forma de uma pilha, precisamos decidir as operações possíveis de serem implementadas sobre esta pilha de dados. As operações mais comuns são definidas a seguir:

- Criar uma pilha vazia;
- Testar se a pilha está vazia;
- Obter o elemento topo da pilha;
- Inserir um elemento na pilha (*push*)
- Retirar um elemento da pilha (*pop*)

Formas de representação

Existem várias formas possíveis de se implementar uma pilha, que se distinguem pela natureza dos seus elementos (tipo do dado armazenado), pela maneira como os elementos são armazenados (estática ou dinamicamente) e pelas operações disponíveis para o tratamento da pilha.

Dessa forma, uma pilha pode ser representada, basicamente, de duas formas:

- Estática: Uma pilha com implementação estática é caracterizada por utilizar uma estrutura estática (vetor) para representar a pilha.
- Dinâmica: Uma pilha com implementação dinâmica é caracterizada por utilizar uma estrutura de dados dinâmica (lista) para representar a pilha.

No caso geral de listas, uma grande vantagem de se utilizar alocação encadeada sobre a seqüencial refere-se à eliminação de deslocamentos na inserção e remoção. No caso das pilhas, as operações de deslocamentos não ocorrem. Sendo assim, numa aplicação onde o número máximo de elementos da pilha é conhecido, a implementação estática é bem simples e vantajosa.

2.2. Pilha Estática

Comumente, em aplicações computacionais que precisam de uma estrutura de pilha, é comum sabermos de antemão o número máximo de elementos que podem estar armazenados simultaneamente na pilha, isto é, a estrutura da pilha tem um limite conhecido.

Dessa forma, a implementação da pilha pode ser feita usando simplesmente um vetor, devido a sua simplicidade, como mostra a figura 2.25.

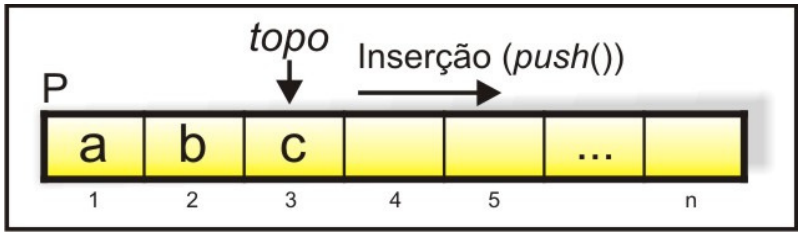


Figura 2.25: Implementando pilha através de um vetor P.

Considerando o vetor P utilizado para armazenar os elementos da pilha. Os novos elementos inseridos na pilha através da operação empilhar (*push*) ocuparão as primeiras posições do vetor. Desta forma, se temos n elementos armazenados na pilha, o elemento n será o elemento do topo.

Vantagens e desvantagens de usar pilhas estáticas

A maioria das aplicações computacionais, comumente a quantidade máxima de elementos possíveis de uma pilha é conhecido a priori. Dessa forma, a implementação desta através de vetores é simples, pois em pilhas não ocorrem operações de deslocamentos no vetor devido às inserções e remoções (inserção e remoção em pilhas obedecem a política LIFO).

Porém, se a quantidade máxima de elementos de uma pilha não for conhecida a priori, a implementação estática torna-se desvantajosa, podendo resultar numa alocação de memória grande demais ou insuficiente para uma determinada aplicação.

Operações sobre pilha estática

Uma pilha estática é definida como um vetor, no qual "*tipo"* é o tipo de dado a ser representado em cada elemento da pilha.

Dessa forma, temos:

```
Constante
        m = 10  //número máximo de elementos da pilha
Tipo
        Pilha = vetor [1..m] de tipo;
```

Em JAVA, podemos definir a classe nó como sendo:

```
class Pilha
{
private int tamanho; //tamanho máximo da pilha
private int[] vetorPilha; //vetor que será armazenado os elementos
private int topo; //referência para o topo da pilha
}
```

A partir da definição da estrutura de dados de uma pilha seqüencial, podemos definir suas duas principais operações: de empilhar (*push*) e desempilhar (*pop*).

a) Operação de empilhar: esta operação é utilizada para inserir um novo elemento no topo da pilha.

```
Algoritmo: Inserção na pilha P
variável P: Pilha; variável topo: inteiro;
topo = 0; //informando que a pilha está vazia
se topo ≠ m então
        topo = topo + 1;
        P[topo] = dado;
senão
        imprima("Pilha cheia");
```

Para a operação de empilhar (*push*), temos o seguinte método em JAVA:

```
public void int push(int dado)
{
vetorPilha[++topo] = dado;
}
```

b) Operação de desempilhar: esta operação é utilizada para remover um novo elemento do topo da pilha.

```
Algoritmo: Remoção na pilha P
variável P: Pilha; variável topo: inteiro;
Retirado: tipo;
se topo ≠ 0 então
        retirado = P[topo];
        topo = topo + 1;
senão
        imprima("Pilha vazia");
```

Para a operação de desempilhar (*pop*), temos o seguinte método em JAVA:

```
public void int pop()
{
Return vetorPilha[topo - -];
}
```

Praticar: utilizando pilhas estáticas, descreva um procedimento para inverter uma seqüência dado. Por exemplo, se a seqüência dada for <a,b,c,d> o seu procedimento deve retornar <d,c,b,a>.

2.3. Pilha Dinâmica

Em aplicações que utilizam pilhas, quando o número máximo de elementos que serão armazenados não é conhecido, devemos implementar a pilha usando uma estrutura de dados dinâmica (empregando listas encadeadas).

Como visto no capítulo anterior, uma lista encadeada é uma estrutura constituída por elementos, conhecidos por nós, que contém um dado qualquer e uma referência para o próximo elemento. Dessa forma, para representar uma pilha através da estrutura lista, basta considerar que a referência para o primeiro elemento da lista seja considerada o topo da pilha, como visto na figura 2.26.

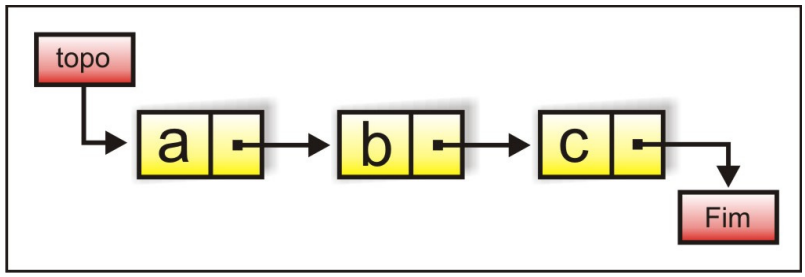


Figura 2.26: Implementando pilha através de lista encadeada.

<u>Vantagens e desvantagens de usar pilhas dinâmicas</u>

Quando em uma determinada aplicação de pilhas não se conhece a priori o tamanho máximo de elementos a ser utilizado, a utilização de estrutura de dados dinâmica lista encadeada é bem interessante, pois à medida que é necessário inserir um novo

elemento basta alocar o espaço de memória necessário para sua inserção.

Porém, como em listas encadeadas, a principal desvantagem da implementação dinâmica de pilhas está no fato de que um elemento da pilha sempre conterá uma referência para o outro elemento da pilha, o que gera uma utilização maior de memória.

Operações sobre pilha dinâmica

As operações em pilhas dinâmicas utilizam os mesmos conceitos de listas encadeadas, porém na pilha a inclusão e a retirada só podem ser realizadas no topo, ou seja, sempre o último nó inserido na pilha.

Dessa forma, podemos definir a estrutura de dados pilha dinâmica como a seguir:

```
Tipo
    Pilha::reg(  dado: tipo;
                 próximo:ref Pilha);
```

Definida a estrutura, podemos desenvolver suas principais operações: de empilhar (*push*) e desempilhar (*pop*).

a) Operação de empilhar: esta operação é utilizada para inserir um novo elemento no topo da pilha.

```
Algoritmo: Inserção na pilha P
variável P, topo: Pilha;
alocar(P);
P↑.dado = dado;
P↑.proximo = topo;
topo = P;
```

A partir deste algoritmo de inserção é possível inserir dinamicamente um novo elemento para a pilha. Note que para a inserção é necessário apenas alocar espaço em memória para o novo elemento (nó). Depois que o espaço é alocado, a referência

próximo aponta para o topo, como pode ser visualizado na figura 2.27.

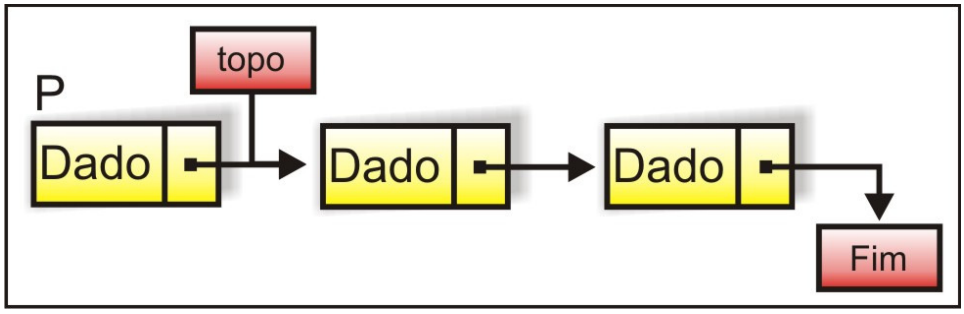


Figura 2.27: Alocando novo nó para ser inserido na pilha.

Logo em seguida, o topo passa a apontar para este novo nó. Dessa forma, o novo nó é inserido no início da lista, tornando-se o novo topo da pilha, como mostra a figura 2.28.

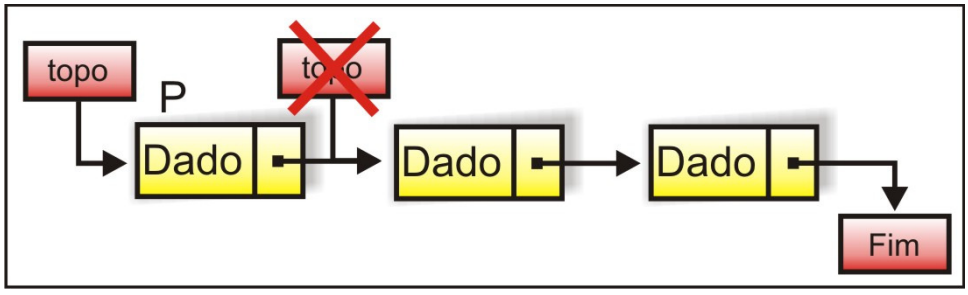


Figura 2.28: Inserção de um novo nó na pilha.

b) Operação de desempilhar: esta operação é utilizada para remover um novo elemento do topo da pilha.

```
Algoritmo: Remoção da pilha P
variável  topo, retirado: Pilha;
se topo ≠ NIL
        retirado = topo;
        topo = topo↑.próximo;
        desalocar(retirado);
senão
        imprima("Pilha vazia");
```

Neste algoritmo, o novo topo passa a referenciar o *próximo* do topo antigo. Em seguida, basta desalocar o espaço de memória do nó retirado, como visualizado na figura 2.29.

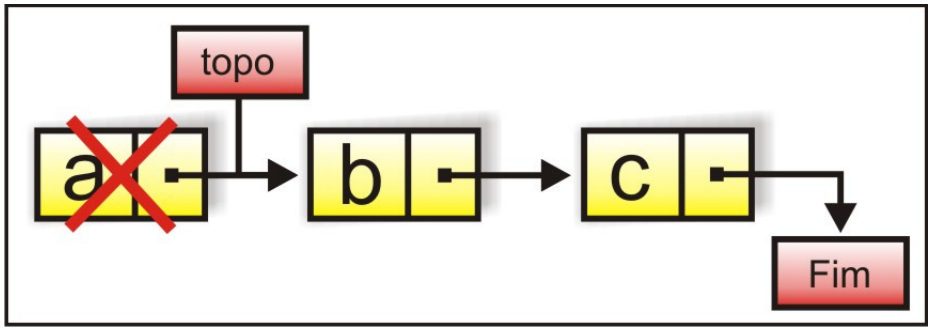


Figura 2.29: Remoção de um nó da pilha.

Praticar: utilizando pilhas dinâmicas, descreva um procedimento para inverter uma seqüência dado. Por exemplo, se a seqüência dada for <a,b,c,d> o seu procedimento deve retornar <d,c,b,a>.

2.4. Exercícios

1. Defina o que é uma Pilha e aponte no que se diferencia de Lista.
2. Aponte as duas principais operações de Pilhas e suas utilidades.
3. Descreva com detalhes as vantagens e desvantagens de se utilizar pilha estática.
4. Descreva com detalhes as vantagens e desvantagens de se utilizar pilha dinâmica.
5. Considerando uma pilha de números inteiros, escreva um programa que forneça o maior, o menor e a média aritmética de seus elementos.
6. Faça um programa que defina as operações *push* e *pop* de uma pilha dinâmica, além de um método para informar qual o elemento do topo

7. Faça um programa que cadastre vários professores (nome, salário e titulação) em uma estrutura de dados do tipo PILHA. Posteriormente o programa deve mostrar:
    a. os nomes dos funcionários que possuem uma determinada titulação fornecida pelo usuário. As possíveis titulações são: Mestre, Doutor e Especialista.
    b. os dados de um determinado professor, cujo nome é fornecido pelo usuário.
8. Escreva um algoritmo que converta uma pilha seqüencial, em uma pilha encadeada. Considere a lista seqüencial com no máximo 100 elementos.
9. Faça um programa que contenha as principais operações da Estrutura de Dados Pilha Estática. Esta pilha deverá ser de números inteiros (tipo do dado da pilha). O seu programa deverá conter as seguintes operações:
    a. criar Pilha Estática;
    b. verificar se a Pilha está vazia;
    c. adicionar um novo elemento na Pilha;
    d. remover um elemento da Pilha;
    e. Informar qual o elemento do topo da Pilha;
    f. Informar a quantidade de elementos da Pilha
    g. Imprimir todos os elementos da Pilha (da base para o topo)
10. Baseado na questão anterior (Questão 9), implemente as operações definidas utilizando uma estrutura de dados Pilha Dinâmica.

UNIDADE I

ESTRUTURAS DE DADOS LINEARES

3. FILAS

3.1. Introdução

Outra estrutura de dados muito utilizada na computação e no nosso dia-a-dia é a fila. Estamos acostumados com as filas em diversos lugares: nos bancos, supermercados, hospitais, entre outros. Sua importância está no fato de determinarem a ordem de atendimento das pessoas, ou seja, em uma fila, sempre o próximo a ser atendido será a primeira pessoa da fila, que normalmente, foi o primeiro a chegar.

Quando a primeira pessoa de uma fila é atendida, a pessoa logo atrás dela torna-se o primeiro da fila. Normalmente, para entrar em uma fila, uma pessoa deve se colocar na última posição (fim da fila). Desta forma, quem chega primeiro tem prioridade de atendimento.

As Filas são estruturas de dados que armazenam os elementos de maneira seqüencial (linear). Da mesma forma da estrutura de dados pilhas, as filas têm suas operações mais restritas do que as operações das listas, possuindo prioridades na remoção de seus elementos.

O que diferencia a fila da pilha é a ordem de saída dos seus elementos. Numa pilha, enquanto que a política de acesso é “o último que entra é o primeiro que sai”, na fila a política de acesso é conhecido como FIFO (*first in first out* – “primeiro a entrar é o primeiro a sair”). A idéia fundamental da fila é que só podemos inserir um novo elemento no final da fila e só podemos retirar o elemento do início, como mostrado na figura 2.30.

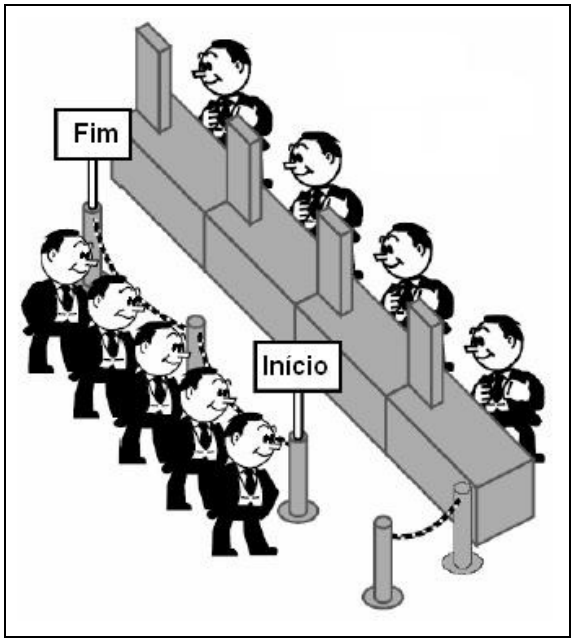


Figura 2.30: Fila de um banco.

DEFINIÇÃO: dada um fila $F = (a_1, a_2, ..., a_n)$, dizemos que a_1 é o elemento do início da fila (saída); a_n é o elemento do fim da fila (entrada); e a_{i+1} está a atrás na fila de a_i.

Operações associadas a filas

Uma vez definido que um conjunto de dados será representado sob a forma de uma fila, precisamos decidir as operações possíveis de serem implementadas sobre esta fila de dados. As operações mais comuns são definidas a seguir:

- Criar uma fila vazia;
- Testar se a fila está vazia;
- Obter o elemento do início da fila;
- Obter o elemento do fim da fila;
- Inserir um elemento no fim da fila;
- Retirar um elemento da fila.

Formas de representação

Como em pilhas, a implementação de filas se distingue pela natureza dos seus elementos (tipo do dado armazenado), pela maneira como os elementos são armazenados (estática ou dinamicamente) e pelas operações disponíveis para o tratamento da fila.

Dessa forma, uma fila também pode ser representada, basicamente, de duas formas:

- Estática: Uma fila com implementação estática é caracterizada por utilizar uma estrutura estática (vetor) para representar a fila.
- Dinâmica: Uma fila com implementação dinâmica é caracterizada por utilizar uma estrutura de dados dinâmica (lista encadeada) para representar a fila.

Pelas suas características, as filas têm as eliminações feitas no seu início e as inserções feitas no seu final. A implementação dinâmica torna mais simples as operações (usando uma lista encadeada com referência para as duas pontas). Já a implementação seqüencial é um pouco mais complexa, mas pode ser usada quando há previsão do tamanho máximo da fila.

3.2. Fila Estática

Como no caso da pilha, as filas podem ser implementadas utilizando uma estrutura de dados estática (vetor). Para isso, devemos definir o número máximo de elementos na fila. Podemos observar que o processo de inserção e remoção em extremidades opostas fará com que a fila se movimente no vetor. Por exemplo, considerando uma fila de um banco, se os indivíduos José, Maria, João e Joana chegarem ao banco, nessa ordem, e depois serem atendidos dois indivíduos, a fila não estará mais nas posições iniciais do vetor. A figura 2.31 ilustra a configuração da fila após a chegada

dos primeiros quatro indivíduos e a figura 2.32 após o atendimento dos dois primeiros indivíduos.

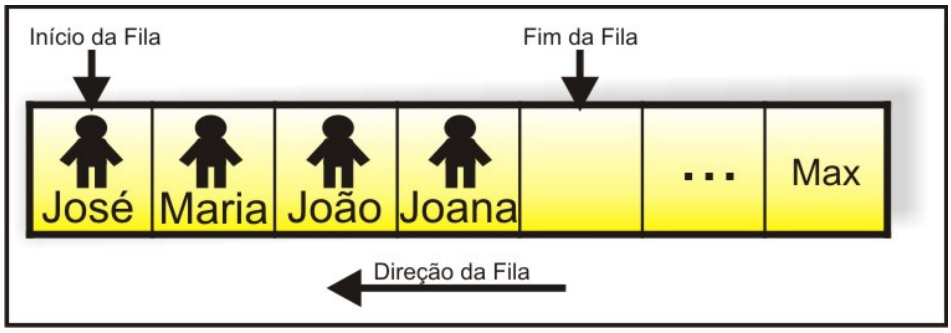


Figura 2.31: Fila representada por vetor após a chegada de 4 indivíduos.

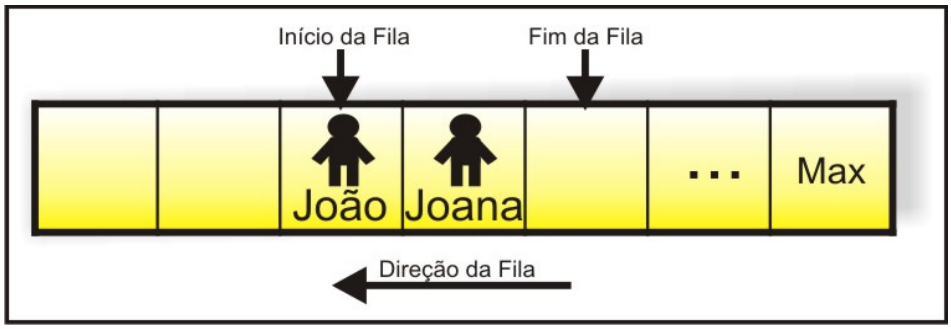


Figura 2.32: Fila representada por vetor após a saída de 2 indivíduos.

Em filas estáticas, facilmente observa-se que em um determinado instante o final do vetor será atingido, devido às inserções. Podemos observar também, que à medida que as pessoas forem atendidas na fila (remoções), o início do vetor passa a ser inutilizado, como mostra a figura 2.33.

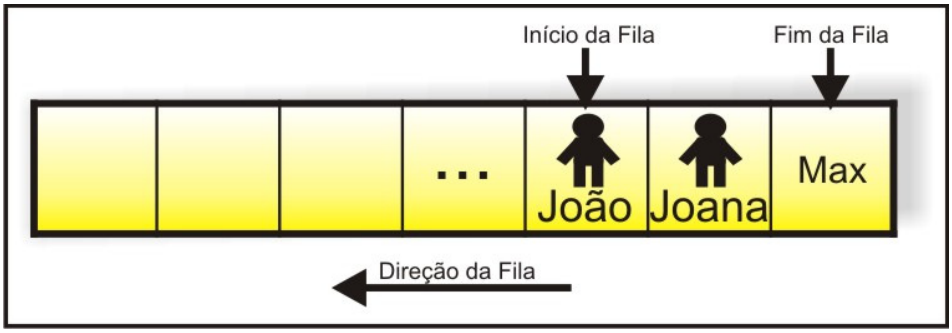


Figura 2.33: Fila através de vetor depois de várias inserções e remoções.

Para podermos reaproveitar as primeiras posições livres do vetor sem precisarmos implementar uma re-arrumação dos elementos (um trabalho árduo), podemos incrementar as posições do vetor de forma circular: se o último elemento da fila ocupa a última posição do vetor, inserimos os novos elementos a partir do início do vetor. Assim, na figura 2.34, poderíamos inserir as novas pessoas no início do vetor.

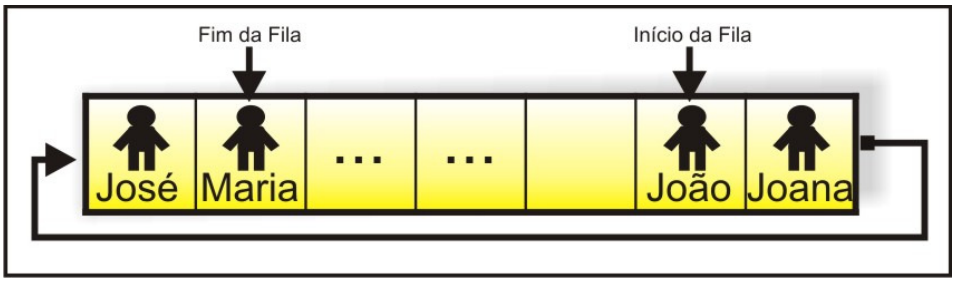


Figura 2.34: Fila circular.

Para este tipo de implementação é preciso de um novo componente para indicar quantos elementos existem na fila no momento.

Vantagens e desvantagens de usar filas estáticas

A implementação estática de filas é vantajosa quando há previsão do tamanho máximo da fila. Porém, se a quantidade máxima de elementos de uma fila não for conhecida a priori, a implementação estática torna-se desvantajosa, podendo resultar numa alocação de memória grande demais ou insuficiente para uma determinada aplicação.

Operações sobre fila estática

Podemos definir uma fila estática como um vetor, no qual "*tipo*" é o tipo de dado a ser representado em cada elemento da fila.

Dessa forma, temos:

```
Constante       m = 10  //número máximo de elementos da pilha
Tipo    Fila = vetor [1..m] de tipo;
```

As filas exigem uma implementação um pouco mais detalhada. São necessários definir duas variáveis, início e fim, que definem o início e o fim da fila, respectivamente. Quando se adiciona um novo elemento na fila, a variável fim é incrementada; de forma análoga, quando se elimina um elemento da fila, a variável início é incrementada.

Note que, após qualquer operação (inserção ou remoção na fila), deve-se sempre ter noção dos valores das variáveis início e fim, para que seja possível saber o local correto de inserção e remoção de elementos da fila.

Como mencionado anteriormente, em um determinado instante o final do vetor será atingido, o que pode dar origem à falsa impressão de memória esgotada (falta de espaço para novas inserções). Como dito, para podermos reaproveitar as primeiras posições livres do vetor sem precisarmos implementar uma re-arrumação dos elementos, podemos incrementar as posições do vetor de forma circular: se o último elemento da fila ocupa a última posição do vetor, inserimos os novos elementos a partir do início do vetor.

O algoritmo a seguir demonstra os procedimentos para inserção de um novo elemento na fila.

```
Algoritmo: Inserção na fila F
variável F: Fila;
variável início,  fim, aux: inteiro;
//valores iniciais para início e fim, apontando que a fila está vazia
início = fim = 0;
aux = (fim mod m) + 1;
se aux ≠ início então
        fim = aux;
        F[fim] = dado;
        se início = 0 então
                início = 1;
senão
        imprima("Fila cheia");
```

Neste algoritmo, podemos observar que antes de inserir um novo elemento, uma variável auxiliar *aux* recebe, provisoriamente, o

valor do resto da divisão da variável *fim* com a quantidade máxima de elementos da fila, adicionado de 1. Este cálculo é realizado a fim de respeitar o comportamento circular da fila.

Como exemplo, consideremos uma fila de banco vazia (início e fim definidos como zero). Ao chegar o primeiro cliente (procedimento de inserção), teremos os seguintes valores:

- aux = início = fim = 1.

Ao chegar o próximo cliente, teremos os seguintes valores:

- aux = 2; início = 1; fim = 2;

Quando a variável *fim* se tornar igual à quantidade máxima permitido na fila (*m*), a variável *aux* terá valor igual a *início*, resultando na notificação de que a fila não contém mais espaço para inserções.

Logo a seguir, podemos visualizar o algoritmo de remoção de um elemento da lista.

```
Algoritmo: Remoção na fila F
variável F: Fila;
variável recuperado: Fila;
variável início,  fim, aux: inteiro;
se início ≠ 0 então
        recuperado = F[início];
        se início = fim então
                início = fim = 0;
        senão
                início = (início mod m) + 1;
senão
        imprima("Fila vazia");
```

Note no algoritmo, que também é feito um teste análogo ao algoritmo de inserção, a fim de permitir inserções no início do vetor, logo após várias remoções, eliminando-se assim à falsa impressão de memória esgotada.

3.3. Fila Dinâmica

As filas também podem ser implementadas dinamicamente, representadas por uma lista simplesmente encadeada, no qual cada nó guarda uma referência para o próximo nó da lista. Como teremos que inserir e retirar elementos das extremidades opostas da lista, que representarão o início e o fim da fila, será preciso usar duas referências, inicio e fim, que apontam, respectivamente, para o primeiro e para o último elemento da fila. A figura 2.35 mostra esta situação.

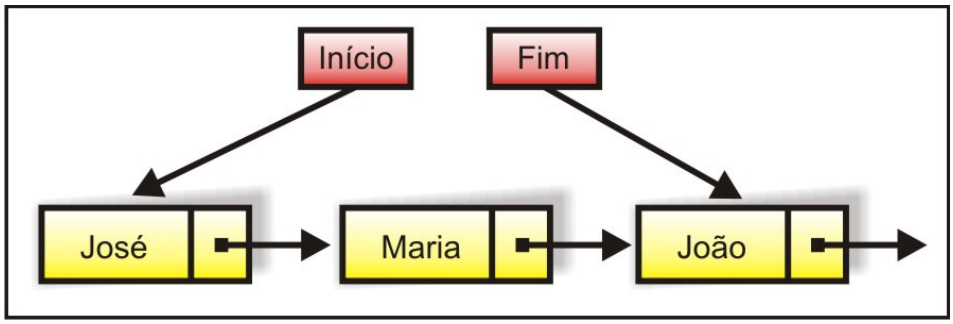


Figura 2.35: Representação de filas através de lista encadeada.

Quando se deseja retirar um elemento da fila (no caso da fila do banco, um cliente será atendido), basta remover o nó do início da fila e fazer com que inicio referencie o sucessor do nó removido. A figura 2.36 ilustra o processo de remoção de um nó da fila.

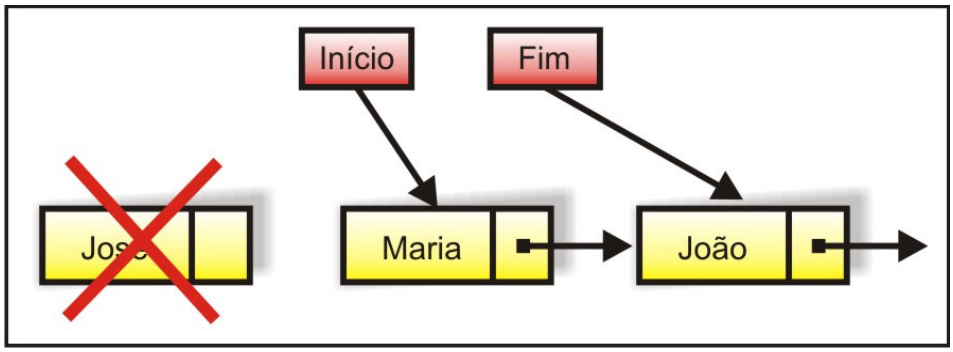


Figura 2.36: Remoção de uma fila dinâmica.

Da mesma forma, para inserir um elemento na fila (no caso da fila do banco, um novo cliente chega), basta acrescentar à lista um sucessor para o último nó, referenciado por fim, e fazer com que fim referencie este novo nó. A figura 2.37 ilustra esse procedimento.

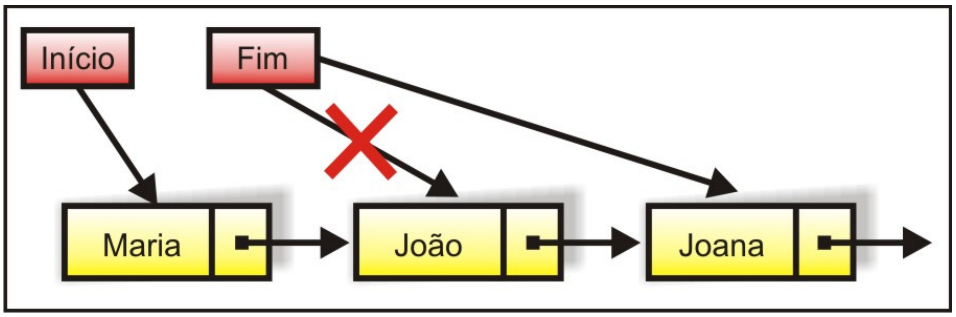


Figura 2.37: Inserção de uma fila dinâmica.

Vantagens e desvantagens de usar filas dinâmicas

Filas dinâmicas são bastante vantajosas quando não se conhece o tamanho máximo da fila. Dessa forma, à medida que desejamos inserir novos elementos na fila, basta alocar um espaço de memória adequado e inserir este novo elemento.

Porém, como em listas encadeadas, a principal desvantagem da implementação dinâmica está no fato de que um elemento da fila sempre conterá uma referência para o outro elemento da pilha, o que gera uma utilização maior de memória.

Operações sobre fila dinâmica

As operações em filas dinâmicas também utilizam os mesmos conceitos de listas encadeadas. Porém, as filas exigem duas variáveis para fazer referência ao início e ao fim da fila.

Dessa forma, podemos definir a estrutura de dados fila dinâmica como a seguir:

```
Tipo
        Fila::reg(    dado: tipo;
                      próximo:ref Fila);
```

Definida a estrutura, podemos desenvolver as operações de inserção e remoção em filas dinâmicas.

a) Operação de inserção: esta operação é utilizada para inserir um novo elemento no fim da fila.

```
Algoritmo: Inserção na fila F
variável F, início, fim: Pilha;
alocar(F);
F↑.dado = dado;
F↑.próximo = NIL;
se fim ≠ NIL então
        fim↑.próximo = F;
senão
        início = F;
fim = F;
```

A partir deste algoritmo de inserção é possível inserir dinamicamente um novo elemento para a fila. Note que para a inserção é necessário alocar espaço em memória para o novo elemento (nó). Depois que o espaço é alocado, o novo nó é inserido no fim da fila (o último elemento passa a referenciar o novo nó, deixando de ser o último elemento) e a variável fim passa a referenciar o novo nó inserido, como pode ser visualizado na figura 2.38.

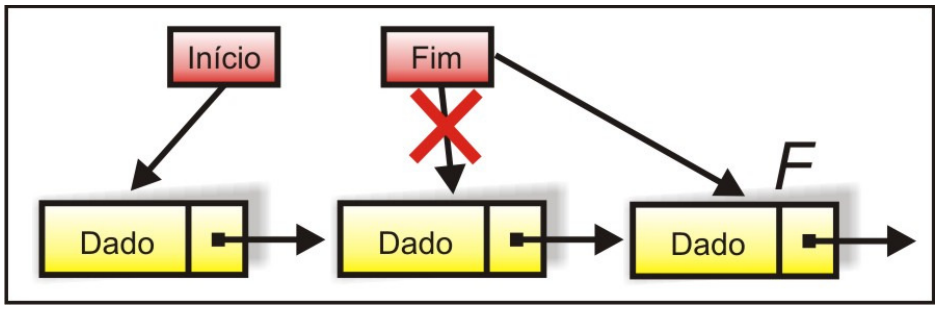


Figura 2.38: Inserindo um novo elemento no fim da fila.

b) Operação de remoção: esta operação é utilizada para remover um elemento do início da fila.

```
Algoritmo: Remoção da fila F
variável F, início, fim: Pilha;
variável recuperado: tipo;
se início ≠ NIL então
        F = início;
        início = início↑.próximo;
        se início = NIL então
                fim = NIL;
        recuperado = F↑.dado;
        desalocar(F);
senão
        imprima("Fila vazia");
```

Note no algoritmo, que se a fila contém elementos, a variável F é definida como o elemento do início da fila e a variável início é redefinida para o próximo elemento da fila. Logo em seguida, o espaço de memória do elemento F (antigo início da fila) é liberado. Este procedimento pode ser visualizado na figura 2.39.

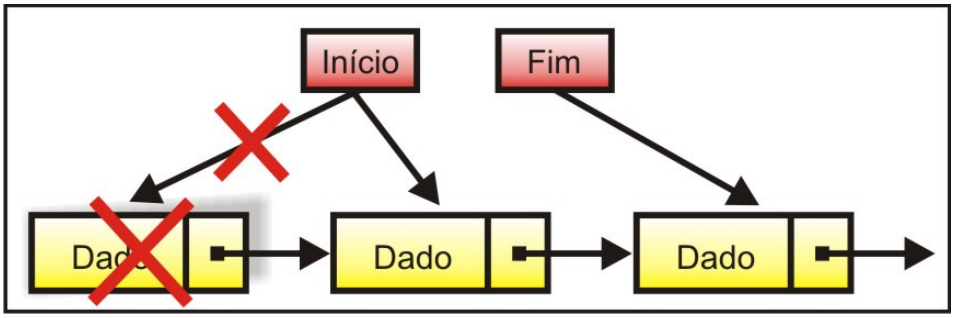


Figura 2.39: Procedimento de remoção de um elemento no início da fila.

3.4. Exercícios

1. Descreva com detalhes as vantagens e desvantagens de se utilizar fila estática.
2. Descreva com detalhes as vantagens e desvantagens de se utilizar fila dinâmica.

3. Considerando uma fila de números inteiros, escreva um programa que forneça o maior, o menor e a média aritmética de seus elementos.
4. Faça um programa para cadastrar 8 funcionários (nome e salário) em uma estrutura de dados do tipo FILA. Posteriormente o programa deve mostrar:
    a. o nome do funcionário que tem maior salário;
    b. mostrar o nome do funcionário que tem menor salário;
    c. a média salarial de todos os funcionários
5. Utilizando as operações conhecidas de inserção e remoção em uma fila, implemente um programa que crie três filas: fila F, Fila F_Impar e a fila F_par. O seu programa deve ler dados e armazená-los na fila F e em seguida separar todos os valores guardados em F de tal forma que os valores pares são movidos para a fila F_par e os valores ímpares são movidos para a fila F_impar.
6. Faça um programa que contenha as principais operações da Estrutura de Dados Fila Estática. Esta fila deverá ser de números inteiros (tipo do dado da fila). O seu programa deverá conter as seguintes operações:
    a. criar Fila Estática;
    b. verificar se a Fila está vazia;
    c. adicionar um novo elemento na Fila;
    d. remover um elemento da Fila;
    e. Informar qual o elemento do início da Fila;
    f. Informar qual o elemento do final da Fila;
    g. Informar a quantidade de elementos da Fila;
    h. Imprimir todos os elementos da Fila (do início para o final).

7. Baseado na questão anterior (Questão 9), implemente as operações definidas utilizando uma estrutura de dados Pilha Dinâmica.

4. WEBLIOGRAFIA

Página da Universidade Aberta do Piauí - UAPI
http://www.ufpi.br/uapi

Página da Universidade Aberta do Brasil- UAB
http://www.uab.gov.br

Página da Secretaria de Educação a Distância do MEC - SEED
http://www.seed.mec.gov.br

Página da Associação Brasileira de Educação a Distância - ABED
http://www.abed.org.br

Algoritmos Animados em JAVA
http://gdias.artinova.pt/projecto/pt/menu_applets.php

Curso de Algoritmos Estruturas de Dados – USP/São Carlos
Profas.: Graça Pimentel, Maria Cristina e Rosane
http://www.icmc.usp.br/~sce182/

Curso de Estruturas de Dados
Instituto de Ciências Matemáticas e de Computação/USP
http://www.icmc.sc.usp.br/manuals/ssce763/

Curso de Estruturas de Dados
Professor Ivan Luiz Marques Ricarte
http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node10.html

5. REFERÊNCIAS BIBLIOGRÁFICAS

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C. , ***Algoritmos***, Ed. Campus, 2002.

DROZDEK, A. , ***Estruturas de Dados e Algoritmos em C++***, Ed. Thomson, 2002.

GOODRICH, M. T. & TAMASSIA, R. , ***Estruturas de Dados e Algoritmos em Java***, Ed. Bookman, 2007.

LAFORE, R. , ***Estruturas de Dados e Algoritmos em Java***, Ed. Ciência Moderna, 2004.

LAUREANO, M. ***, Estrutura de Dados com Algoritmos em C***, Ed. Brasport, 2008.

LORENZI , F. & MATTOS , P. N. & CARVALHO, T. , ***Estruturas de Dados***, Ed. Cengage Learning, 2006.

PEREIRA***, Estruturas de Dados Fundamentais: Conceitos e Aplicações***, Ed. Érica, 2006.

PREISS, B. R. , ***Estruturas de Dados e Algoritmos***, Ed. Campus, 2000.

PUGA, S. & RISSETI, G. , ***Lógica de Programação e Estruturas de Dados***, Ed. Prentice-Hall, 2004.

RANGEL, J. L. & CERQUEIRA, R. & CELES, W. , ***Introdução a Estrutura de Dados***, Ed. Campus, 2004.

SCHILDT, H. , ***C Completo e Total***, Ed. Makron Books, 1996.

SILVA, O. Q. , ***Estruturas de Dados e Algoritmos Usando C: Fundamentos e Aplicações***, Ed. Ciência Moderna, 2007.

SZWARCFITER, J. L. & MARKENZON, L. , ***Estruturas de Dados e Seus Algoritmos***, Ed. LTC, 1994.

TENEMBAUM, A. M. ***, Estruturas de Dados Usando C***, Ed. Makron Books, 1995.

VELOSO, P. A. S. , ***Estruturas de Dados***. Ed Campus, 1983.

WIRTH, N. , ***Algoritmos e Estruturas de Dados***, Ed. LTC, 1989.

ZIVIANI, N. . ***Projeto de Algoritmos com implementação em PASCAL e C***, Ed. Thomson, 2005.

Nesta unidade examinaremos uma estrutura de dados útil para várias aplicações: a árvore. A importância das estruturas listas, filas e pilhas é inegável, mas elas não são adequadas para representarmos dados que devem ser dispostos de maneira hierárquica o que justifica fortemente a utilização das árvores já que sua principal característica é manter a relação de hierarquia entre seus componentes.

Como acontece com as listas, filas e pilhas, trataremos as árvores basicamente como estruturas de dados em vez de como tipos de dados. Definiremos várias formas dessa estrutura de dados e mostraremos como elas podem ser representadas. Focalizaremos neta unidade basicamente a implementação das estruturas, e não as definições matemáticas relacionadas.

Esperamos que ao final o leitor tenha se convencido da utilidade dos conceitos e processos apresentados, mas guardamos o desejo que ele aprofunde-se cada vez mais nesta imensidão que são as estruturas de dados – árvore.

Cada capítulo é acompanhado de exercícios, sem a solução, preferimos deixar o prazer desta tarefa ao leitor. A bibliografia e a webliografia ao fim das notas são mais do que suficiente para adquirir um conhecimento razoável sobre árvores.

SUMÁRIO

UNIDADE II – ÁRVORES

UNIDADE II
ÁRVORES

1. ÁRVORES GENÉRICAS

1.1. Introdução

Nas unidades anteriores apresentamos estruturas de dados lineares. A importância dessas estruturas é inegável, mas elas não são adequadas para representarmos dados que devem ser dispostos de maneira hierárquica. Exemplificando, temos a estrutura hierárquica de diretórios do nosso computador que pode ser visualizada na ilustração da figura 3.1.

Figura 3.1: Estrutura de diretórios Windows XP.

Além das listas, filas e pilhas vistas anteriormente, existem outras formas usadas para representar os dados, que são chamadas genericamente de não-lineares. Essa representação permite que sejam feitos outros tipos de relações entre os dados.

No caso de árvores, objetivo de estudo desta unidade, a relação existente entre os dados (denominados nós ou nodos) é uma relação de hierarquia ou de composição, onde um conjunto de dados é hierarquicamente subordinado a outro.

Um exemplo bem conhecido de relação de estruturação em árvore é a estruturação de um livro, que é subdividido em capítulos, onde cada capítulo pode ser subdividido em seções, as quais podem possuir tópicos e sub-tópicos associados.

A figura 3.2 mostra a estrutura de um livro que possui um capítulo intitulado Árvores, que por sua vez possui três seções: Introdução, Árvores Binárias e Árvores AVL. Além disso, a seção Árvores Binárias possui uma subdivisão em três tópicos: Conceitos, Percurso e Balanceamento.

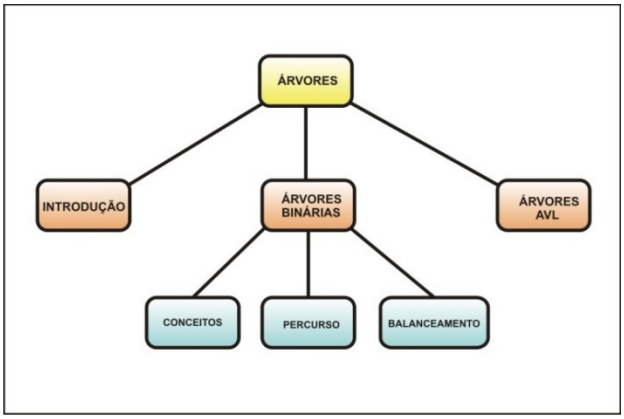


Figura 3.2: Estrutura hipotética de um livro.

Em suma, uma árvore é um tipo abstrato de dados que armazena elementos de maneira hierárquica. Com exceção do elemento do topo, cada elemento da árvore tem um elemento pai e zero ou mais elementos filho. Normalmente o elemento do topo é chamado de raiz.

Em outras palavras, uma árvore é formada por um conjunto finito *T* de elementos denominados vértices ou nós de tal modo que:

- Se T = 0 a árvore é vazia, caso contrário temos um nó especial chamado raiz da árvore (r);

- os demais nós formam m ≥ 1 conjuntos disjuntos $S_1, S_2, ..., S_m$*, onde cada um destes conjuntos é uma árvore. As árvores* S_i *(1 ≤ i ≤ n) recebem denominação de sub-árvores;*

A forma convencional de representar uma árvore está indicada na figura 3.3. Esta árvore possui nove nós *A, B, C, D, E, F, G, H, I, J*, sendo *A* o nó raiz.

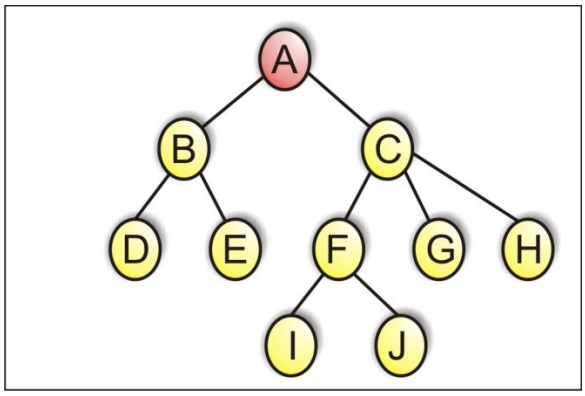


Figura 3.3: Representação gráfica de uma Árvore.

Para fortalecer a definição de árvore, anteriormente apresentada, é importante salientar que os demais nós que a compõe formam conjuntos disjuntos, ou seja, que não possuem elementos em comum (interseção vazia), logo, a estrutura apresentada na figura 3.4 não representa uma árvore.

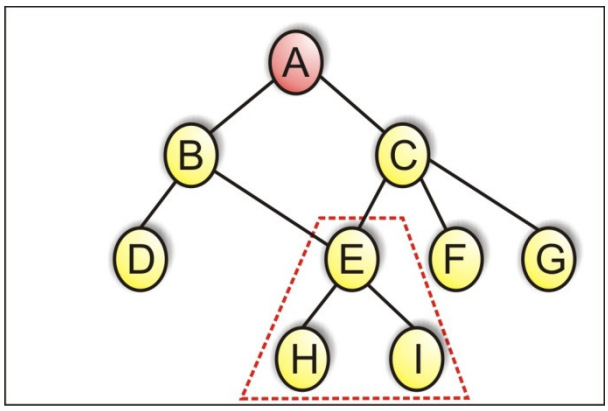


Figura 3.4: Estrutura que não é uma Árvore.

Praticar: As estruturas apresentadas nas figuras 1 e 2 são árvores? Utilize a definição para justificar sua resposta.

1.2. Conceitos básicos

Uma vez entendido a definição de árvore, apresentaremos agora outros conceitos básicos relacionados.

Para facilitar o entendimento, considere o seguinte exemplo de árvore genealógica apresentado na figura 3.5.

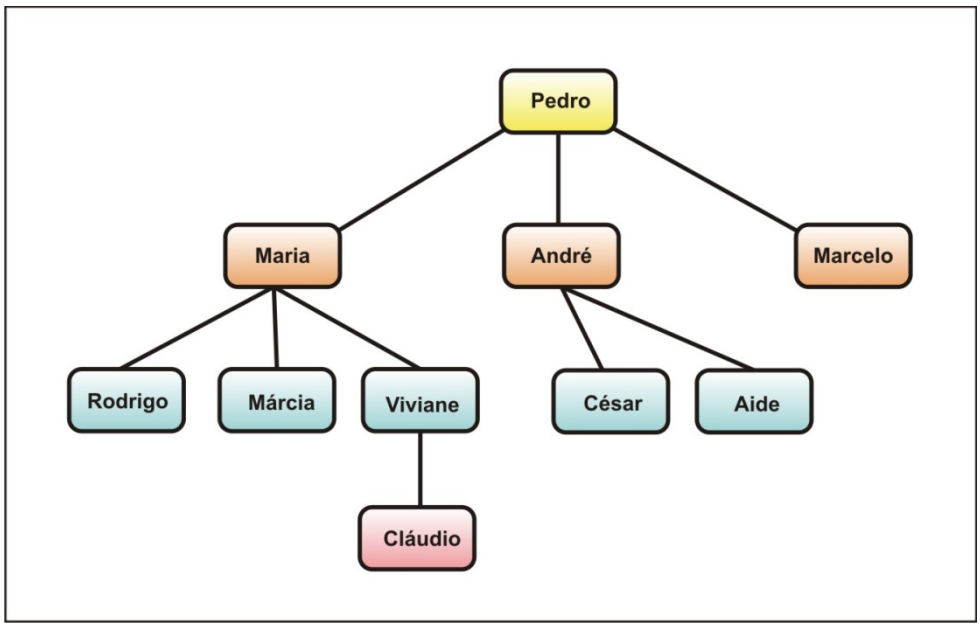


Figura 3.5: Árvore genealógica.

Analisando-a, podemos concluir que:

· Essa árvore possui 10 nós distribuídos aleatoriamente. O nodo Pedro é a **raiz** da árvore, que tem 3 **sub-árvores** com Maria, Andre e Marcelo como raízes;

· O número de sub-árvores de um nó determina o **grau** desse no. Dessa forma, Pedro e Maria têm grau 3, enquanto André tem grau 2 e Marcelo tem grau zero. Nodos que tem grau zero são denominados terminais ou **folhas**;

· Considerando que cada nó tem um grau máximo (numero de sub-árvores das quais esse nó é raiz) e que todos os nós possuem o mesmo grau máximo, definimos esse grau como o **grau da árvore**;

· Uma **árvore** é **homogênea** quando todos os seus nodos possuem as mesmas características de conteúdo, ou seja, o mesmo tipo de informação. Caso contrário ela é então chamada de **árvore heterogênea**;

· Para identificar os nós da estrutura, usamos as denominações da relação de hierarquia existente em uma árvore genealógica. Dessa forma, Pedro é **pai** de Maria, André e Marcelo que são irmãos entre si. No sentido inverso, Rodrigo, Márcia e Viviane são **filhos** de Maria e netos de Pedro. Para conhecermos os **antepassados** de um nodo, basta identificarmos todos os nodos ao longo do caminho entre a raiz e este nodo. Ex: os **antepassados** de Claudio são, na ordem: Pedro, Maria e Viviane;

· Um conceito importante no estudo de árvores e o conceito de **nível**, que representa a distância do nodo até a raiz. Por definição, a raiz da árvore tem nível 0. Na figura 3.5, os nodos Maria, André e Marcelo têm nível 1, os nodos Rodrigo, Márcia, Viviane, Cezar e Aide têm nível 2, assim por diante. O nodo de maior nível nos fornece a **altura (ou profundidade)** da árvore.

Praticar: Qual o grau de Viviane? No exemplo da figura 3.5 a árvore possui que altura?

Observem que vários conceitos fundamentais para o entendimento de árvores são facilmente verificados a partir de uma árvore genealógica. Vamos agora formalizar os conceitos já apresentados bem como outros conceitos relacionados às árvores.

Árvore: Uma árvore *T* é um conjunto finito de elementos denominados nós tais que:

- $T = \phi$, e a árvore é dita vazia, ou
- existe um nó especial *r*, chamado raiz de *T*; os restantes constituem um único conjunto vazio ou são divididos em $m \geq 1$ conjuntos disjuntos não vazios que são as sub-árvores de *r*, cada qual, por sua vez, uma árvore.

Sub-árvore: Se *v* é um nó de *T*, então a notação *Tv* indica a sub-árvore de *T* com raiz em v.

Nós filhos, pais, tios, irmãos e avô: Seja *v* o nó raiz da sub-árvore *Tv* de *T*:

- os nós raízes $v_1, v_2, \ldots v_j$ das sub-árvores de *Tv* são chamados filhos de *v*;
- o nó *v* é chamado pai de $v_1, v_2, \ldots v_j$;
- os nós $v_1, v_2, \ldots v_j$ são irmãos;
- se *z* é filho de v_1 então v_2 é tio de *z* e *v* é avô de *z*;

Grau de saída, descendente e ancestral: O número de filhos de um nó é chamado grau de saída do nó. Se *x* pertence à sub-árvore *Tv*, então, *x* é descendente de *v* e *v* é ancestral, ou antecessor, de *x*.

Nó folha e nó interior: Um nó que não possui descendentes próprios é chamado de nó folha, ou seja, um nó folha é aquele com grau nulo. Um nó que não é folha (isto é, possui grau diferente de zero) é chamado nó interior ou nó interno.

Grau de uma árvore: O grau de uma árvore é o máximo entre os graus de seus nós.

Floresta: Uma floresta é um conjunto de zero ou mais árvores.

Caminho, comprimento do caminho: Uma seqüência de nós distintos $v_1, v_2, ..., v_k$, tal que existe sempre entre nós consecutivos (isto é, entre v_1 e v_2, entre v_2 e v_3, ... , $v_{(k-1)}$ e v_k) a relação "é filho de" ou "é pai de" é denominada um caminho na árvore. Diz-se que v_1 alcança v_k e que v_k é alcançado por v_1. Um caminho de v_k vértices é obtido pela seqüência de k-1 pares. O valor k é o comprimento do caminho.

Nível (ou profundidade) e altura de um nó: O nível ou profundidade, de um nó *v* é o número de nós do caminho da raiz até o nó *v*. O nível da raiz é, portanto, 1. A altura de um nó *v* é o número de nós no maior caminho de v até um de seus descendentes. As folhas têm altura 1.

Nível da raiz (profundidade) e altura de uma árvore: A altura de uma árvore *T* é igual ao máximo nível de seus nós. Representa-se a altura de *T* por *h(T)* e a altura da sub-árvore de raiz *v* por *h(v).*

Árvore Ordenada: Uma árvore ordenada é aquela na qual os filhos de cada nó estão ordenados. Assume-se ordenação da esquerda para a direita. A figura 3.6 apresenta uma árvore ordenada em (a) e uma não ordenada em (b), considerando o conteúdo dos seus nós como sendo as letras do nosso alfabeto.

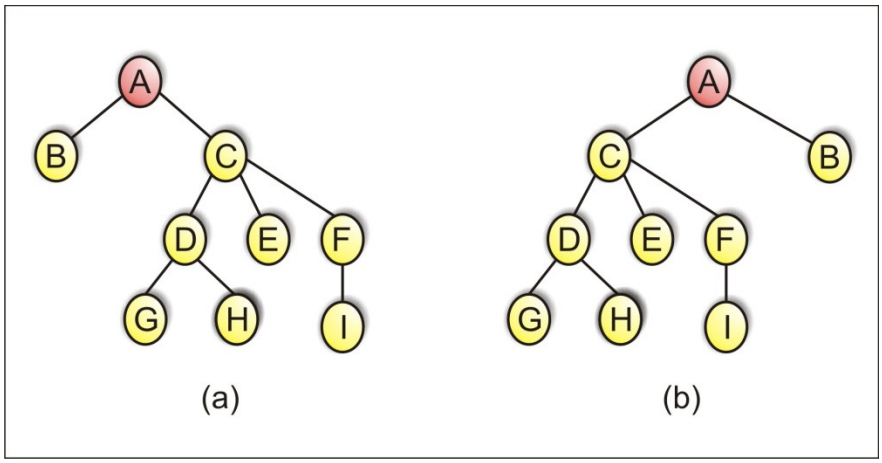


Figura 3.6: Árvores ordenadas e desordenadas.

Árvores Isomorfas: Duas árvores não ordenadas são isomorfas quando puderem se tornar coincidentes através de uma permutação na ordem das sub-árvores de seus nós. Um exemplo de árvores isomórficas pode ser verificado na figura 3.7, pois caso seja permutando as sub-árvores com raiz em B e C as árvores se tornarão coincidentes.

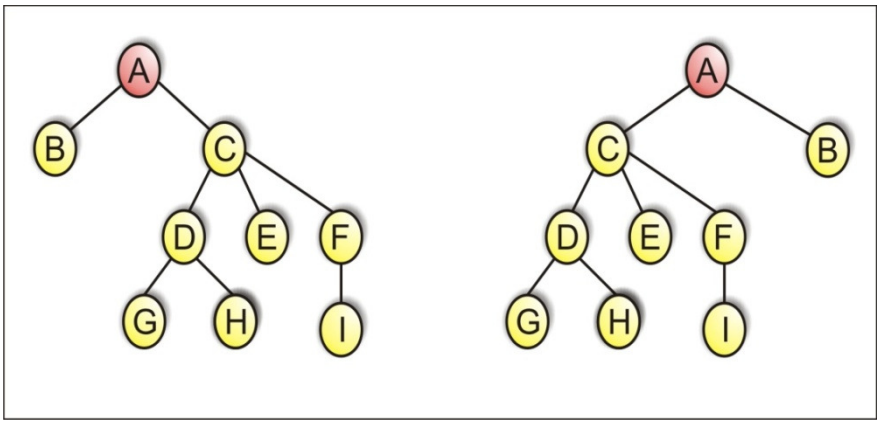


Figura 3.7: Árvores isomorfas.

Árvore Cheia ou Completa**:** Árvore com número máximo de nós. Uma árvore de grau d tem número máximo de nós se cada nó, com exceção das folhas, tem grau d. A figura 3.8 (a) mostra uma árvore cheia de grau 2, pois todos os nós não folhas também possuem grau 2. Porém, na figura 3.8 (b) é possível observar que a árvore não é cheia, pois o nó C possui apenas grau 1 e o grau da árvore é 2.

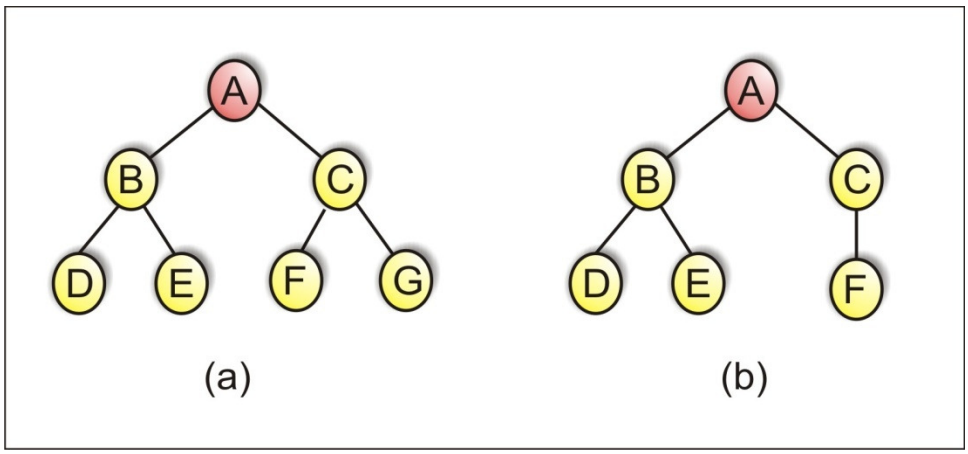


Figura 3.8: Árvore completa e árvore incompleta.

Praticar: Verifique se as árvores da figura 3.3 são completas. Elas são ordenadas? Utilize as definições para justificar sua resposta.

O resumo dos conceitos apresentado é exibido na figura 3.9.

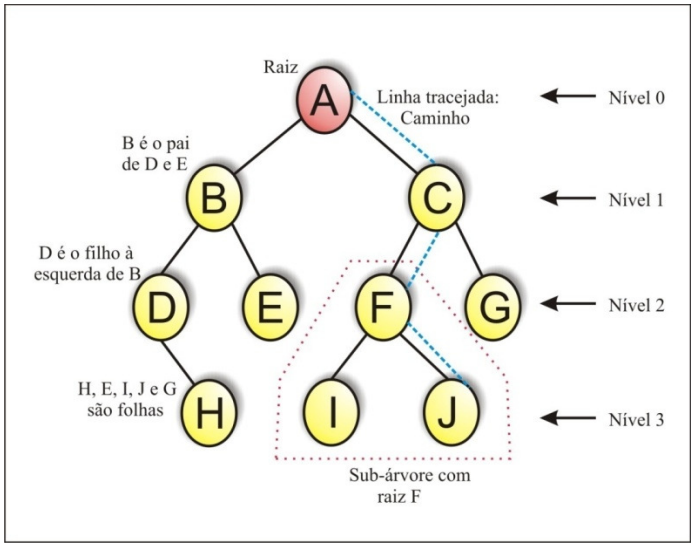


Figura 3.9: Termos em árvore.

1.3. Representação

Existem muitas formas de representar graficamente uma árvore, além das apresentadas anteriormente. Como exemplos temos:

- Representação Natural;
- Representação por Endentação;
- Representação por Conjuntos;
- Representação por Nível.

Representação Natural: o nó raiz ocupa a parte inferior da árvore e os demais estão acima dele, como a estrutura natural das árvores;

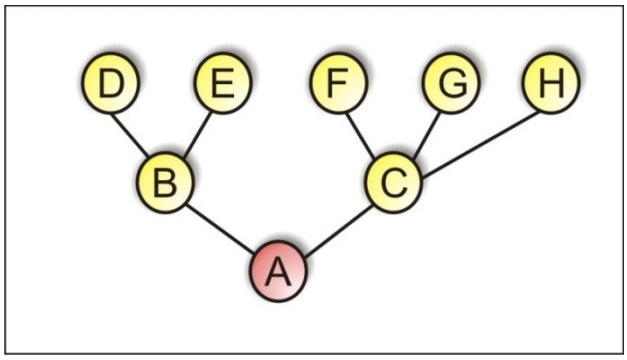


Figura 3.10: Árvores: Notação Natural.

Representação por Endentação: é uma forma utilizada para representar árvores por barras. Sua estrutura lembra o sumário de livros;

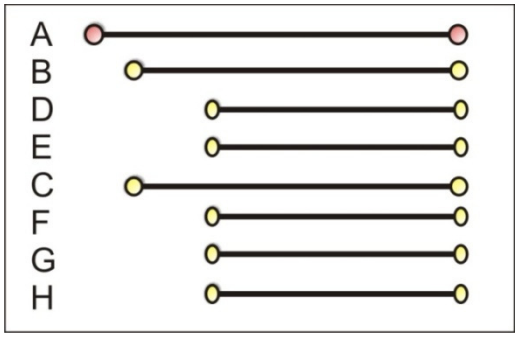


Figura 3.11: Árvores: Notação por Endentação ou Barras.

Representação por Conjuntos: é a notação utilizada para representar árvores como conjuntos aninhados;

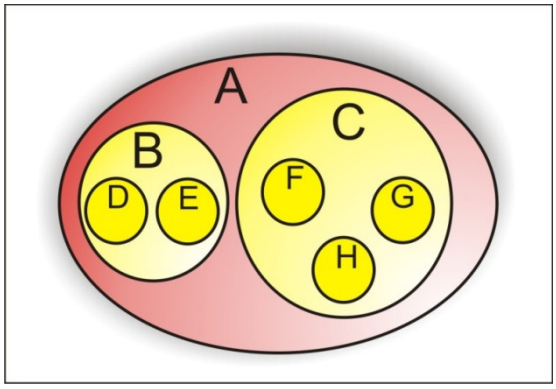


Figura 3.12: Árvores: Notação por Conjuntos.

Representação por Nível: nesta notação é utilizado a idéia de parênteses alinhados e a seqüência de parênteses representa a relação entre os nós da estrutura;

(A (B (D) (E)) (C (F) (G) (H)))

Praticar: Represente as árvores da figura 3.6 em todas as notações apresentadas.

1.4. Alocação

À semelhança do que acontecem com listas lineares, as árvores podem ser alocadas na memória de um computador por adjacência ou por encadeamento.

Alocação por Adjacência: Nesta modalidade, os nós da árvore são representados seqüencialmente na memória, de acordo com uma determinada ordem em que eles aparecem na árvore. A figura 3.13 mostra a alocação seqüencial de uma árvore.

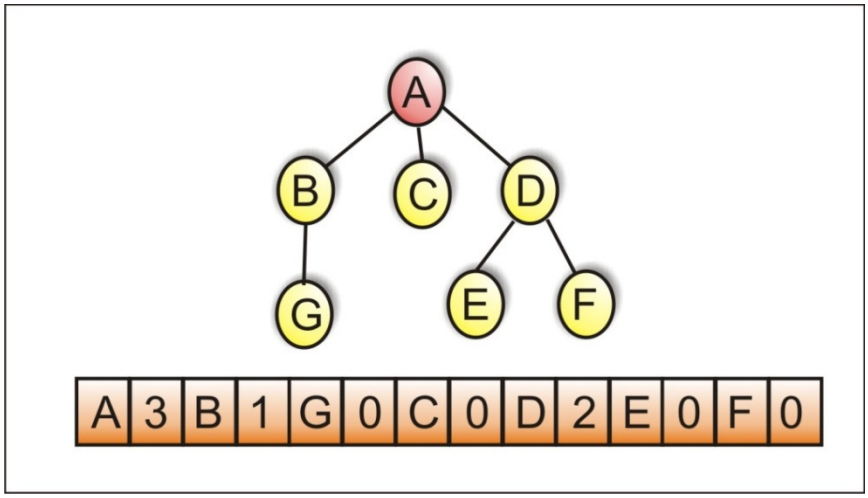


Figura 3.13: Alocação por adjacência.

Os números que aparecem à direita da informação de cada nó na alocação seqüencial indicam o grau do nó correspondente.

A representação na forma de uma lista linear não é indicada em duas situações:
- quando for necessário efetuar operações de inserção e retirada de nodos, pois potencialmente exigira uma grande movimentação dos dados para conseguir o efeito desejado; e
- quando a árvore afasta-se mais e mais da condição de completa, ou seja, não é indicada em situações onde a estrutura de dados é freqüentemente alterada.

Esta forma de alocação é útil, entretanto, quando os nós da árvore devem ser processados na mesma ordem em que aparecem na alocação seqüencial ou especialmente quando as árvores são completas, pois resulta na utilização máxima do espaço alocado.

Alocação Encadeada: A alocação por encadeamento oferece, de um modo geral, uma alternativa mais adequada para a representação de árvores, permitindo, com igual facilidade, manipulação das árvores, bem como diversas ordens de acesso aos nós.

Na alocação encadeada, cada nó é um dado que pode ser alocado dinamicamente e possui espaço para representar tanto a informação do nó como as referências das sub-árvores daquele nó. O principal problema deste esquema é devido ao fato de cada nó poder possuir um número diferente de sub-árvores.

Na seção seguinte, Estruturas, é apresentado um detalhamento desta técnica de alocação bem como suas possíveis formas de implementação.

1.5. Estruturas

Dependendo da aplicação, podemos usar várias estruturas para representar árvores, levando em consideração o número de filhos que cada nó pode apresentar. Se soubermos, por exemplo, que numa aplicação o número máximo de filhos que um nó pode apresentar é 3, podemos montar uma estrutura com 3 campos apontadores para os nós filhos.

Considere a figura 3.14 que representa uma árvore onde cada um dos seus nós possui no máximo três filhos.

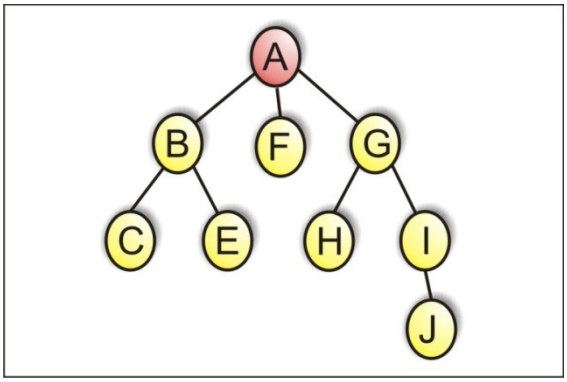


Figura 3.14: Árvore com no máximo dois filhos.

A figura 3.15 apresenta a organização desta estrutura utilizando alocação encadeada.

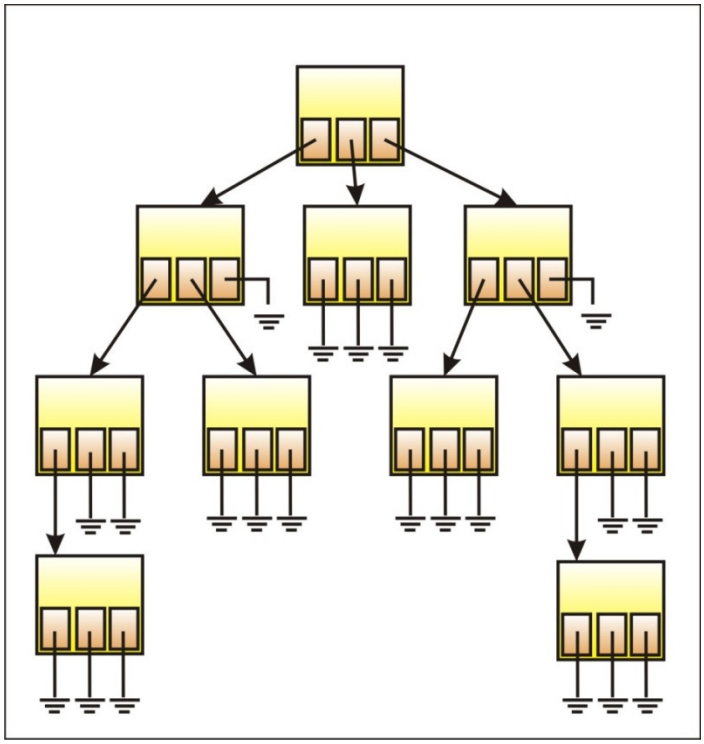


Figura 3.15: Árvores: Notação por Conjuntos.

Como se pode ver no exemplo, em cada um dos nós que tem menos de três filhos, o espaço correspondente aos filhos inexistentes é desperdiçado. Além disso, se não existe um limite superior no número de filhos, esta técnica pode não ser aplicável. O mesmo acontece se existe um limite no número de nós, mas esse limite será raramente alcançado, pois estaríamos tendo um grande desperdício de espaço de memória com os campos não utilizados.

Uma solução que leva a um aproveitamento melhor do espaço utiliza uma “lista de filhos”: um nó aponta apenas para seu primeiro filho, e cada um de seus filhos, exceto o último, aponta para o próximo irmão.

A figura 3.16 mostra o mesmo exemplo representado de acordo com esta nova estruturação.

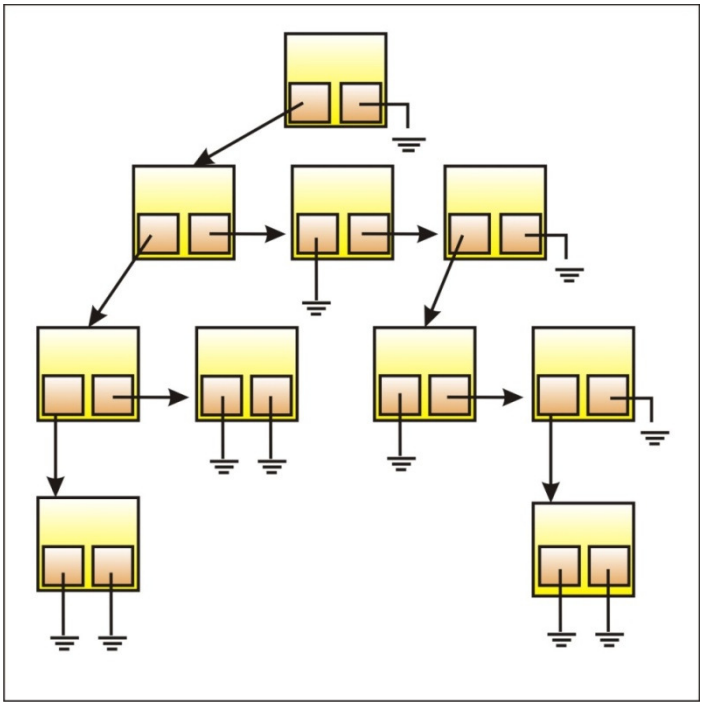


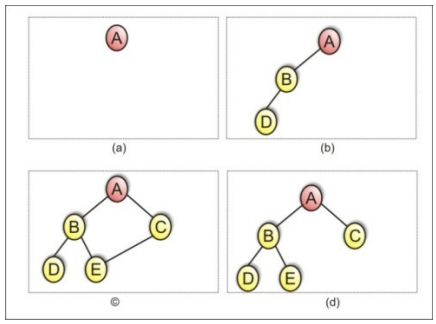
Figura 3.16: Árvores: Notação por Conjuntos.

Uma das vantagens dessa representação é que podemos percorrer os filhos de um nó de forma sistemática, de maneira análoga ao que fizemos para percorrer os nós de uma lista simples.

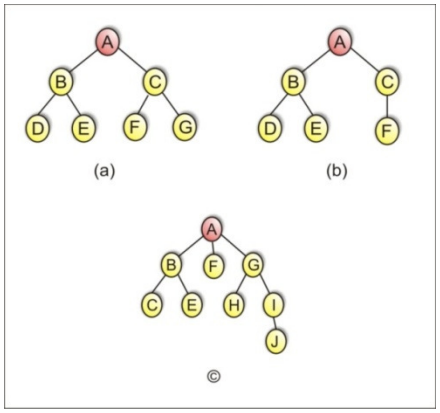
Praticar: Utilizando como referência as árvores da figura 3.8 monte as estruturas de dados que possam representá-las e esboce graficamente como elas seriam representadas utilizando alocação por adjacência e alocação por encadeamento.

1.6. Exercícios

1. Quais das estruturas abaixo são árvores? Justifique sua resposta.



2. Considerando as árvores abaixo responda:

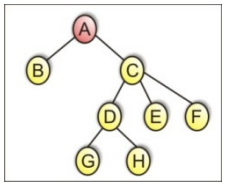


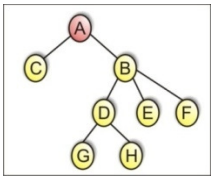
   a. Quantas sub-árvores ela contém?
   b. Quais os nós folhas?
   c. Qual o grau de cada nó?
   d. Qual o grau da árvore?
   e. Quais os ancestrais dos nós B e F?
   f. Identifique todas as relações de parentesco entre os nós.
   g. A árvore é ordenada?
   h. A árvore é completa?

3. Represente na forma natural, por endentação, por conjuntos e por nível cada árvore da questão anterior.

4. Construa uma árvore com base nas informações abaixo:
   - O nó C tem grau 3;
   - O nó X e neto de C e filho de B;
   - O avô de B e A;
   - O nó A tem altura 0 e T tem altura 1;
   - Os antepassados de P são A, T e K, que são também antepassados de H;
   - T tem grau 2, e um dos seus filhos e o nodo S;
   - O nodo G tem 2 sub-árvores que são netos de C;
   - D e irmão de G, e é uma folha;
   - E e F tem graus 0 e 1 respectivamente e são também netos do nodo C;
   - O nodo N tem nível 4.

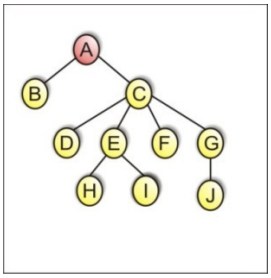
5. As árvores abaixo são isomorfas? Justifique sua resposta.





6. Mostre a representação por adjacência e por encadeamento para as árvores da questão anterior.

7. Considere uma árvore que possui no máximo quatro filhos como a apresentada na figura abaixo. Apresente, em Portugol e em Java estruturas de dados que possa representá-la e mostre graficamente como ela seria representada utilizando alocação por adjacência e alocação por encadeamento.



UNIDADE II
ÁRVORES

2. ÁRVORES BINÁRIAS

2.1. Definição

Uma árvore binária se caracteriza pelo fato de todos os seus nós terem no máximo duas sub-árvores, ou seja, é uma árvore de grau 2. As duas sub-árvores de cada nó são denominadas sub-árvore esquerda e sub-árvore direita.

Em outras palavras, uma árvore binária pode ser definida como um conjunto finito de nós, que é vazio, ou consiste de um nó raiz e dois conjuntos disjuntos de nós, a sub-árvore esquerda e a sub-árvore direita.

A figura 3.17 a seguir ilustra uma estrutura de árvore binária. Os nós *A, B, C, D, E, F* formam uma árvore binária da seguinte maneira: a árvore é composta do nó *A*, da sub-árvore à esquerda formada por *B* e *D*, e da sub-árvore à direita formada por *C*, *E* e *F*. O nó *A* representa a raiz da árvore e os nós *B* e C as raízes das sub-árvores. Finalmente, os nós *D*, *E* e *F* são folhas da árvore.

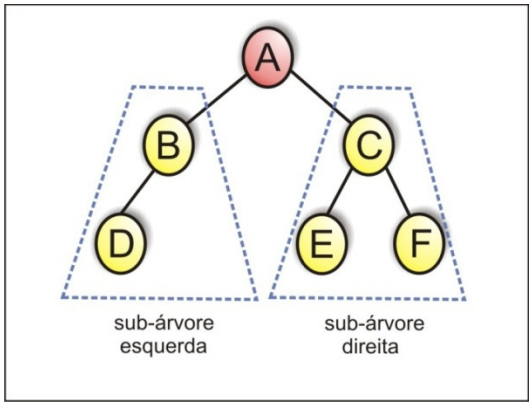


Figura 3.17: Árvore Binária.

Tomemos agora as árvores da figura 3.18. Nela temos duas árvores binárias, sendo (a) completa e (b) incompleta. Geralmente nos referimos aos filhos de um nó em uma árvore binária como sendo da esquerda ou da direita, de acordo com o posicionamento dele. Por exemplo, na figura 3.18 (a) o nó B é filho a esquerda de A e o nó C é filho a direita de A.

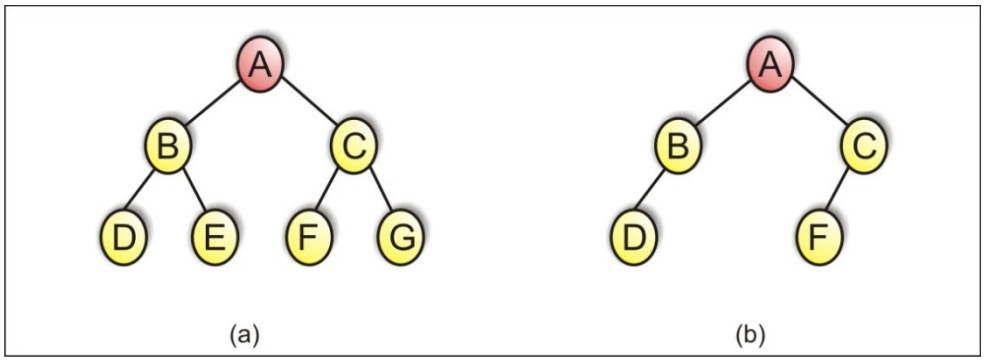


Figura 3.18: Árvores binárias: (a) completa e (b) incompleta

É importante salientar que uma árvore binária não é um caso especial de árvore e sim um conceito completamente diferente. Por exemplo, considere a figura 3.19, note que são duas árvores idênticas, mas são duas árvores binárias diferentes. Isto porque uma delas tem a sub-árvore da direita vazia e a outra a sub-árvore da esquerda.

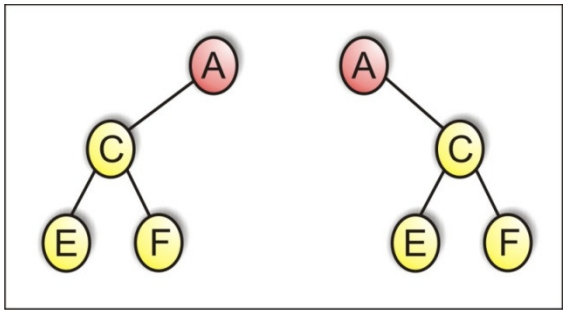


Figura 3.19: Árvores binárias distintas.

Árvores binárias podem ser vistas em diversas situações do cotidiano. Por exemplo, um torneio de futebol eliminatório, do tipo das copas dos países, como a Copa do Brasil, em que a cada etapa os times são agrupados dois a dois e sempre são eliminados metade dos times, é uma árvore binária.

Praticar: Verifique quais das árvores apresentadas no capítulo anterior são binárias.

2.2. Estrutura de dados

Para definirmos uma árvore binária, assim como uma árvore genérica, primeiro precisamos de uma tipo/classe de objetos Nós. Esses tipos/objetos contém dados que representam os objetos sendo armazenados também referências para cada um dos filhos de um nó.

Um tipo de dado que representa um nó passa, então, a ter a seguinte definição:

```
tipo nó :: reg(esquerda:ref nó;
               informação: info;
               direita:ref nó)
```

Em Java, a classe nó poderia ser definida da seguinte forma:

```
class No
{
  int   iDado;          // dado usado como valor chave
  double   fDado;       // outro dado
  No   filhoDireita;    // filho à esquerda
  No   filhoEsquerda; // filho à direita
}
```

Também precisamos de uma classe a partir da qual pode-se instanciar a árvore propriamente dita: o objeto que mantém todos os nós. Chamaremos essa classe de Arvore. Ela tem apenas um campo: uma variável No que armazena a raiz. Ela não precisa de campos para os outros nós porque eles são todos acrescentados a partir da raiz. A classe Arvore tem vários métodos. Eles são usados para localizar, inserir e eliminar nós.

```
class Arvore
{
private  No raiz;                          // único campo de dado em Arvore
public  void  localizar(int chave){ }      // localizar um elemento em Arvore
public  void  inserir(int id, double dd ){ } // inserir um elemento em Arvore
public  void  deletar(int id){ }           // apagar um elemento em Arvore
}
```

A implementação dos métodos da classe Arvore serão vistos mais adiante.

2.3. Percurso

Por percurso em árvores entende-se o ato de percorrer todos os nós da árvore, com o objetivo de consultar ou alterar a informação nelas contida. Existem vários métodos de caminhamento em árvores, que permitem percorrê-la de forma sistemática e de tal modo que cada nó seja visitado apenas uma vez.

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

Uma árvore é uma estrutura não seqüencial, diferentemente de uma lista, por exemplo. Não existe ordem natural para percorrer árvores e, portanto, podemos escolher diferentes maneiras de percorrê-las. Nós iremos estudar três métodos para percorrer árvores. Todos estes três métodos podem ser definidos recursivamente e se baseiam em três operações básicas: visitar a raiz, percorrer a sub-árvore da esquerda e percorrer a sub-árvore da direita. A única diferença entre estes métodos é a ordem em que estas operações são executadas.

Os três métodos são:

- Percurso Pré-Ordem;

- Percurso In-Ordem;

- Percurso Pós-Ordem;

Percurso Pré-Ordem: O primeiro método, conhecido como percurso em pré-ordem, implica em executar recursivamente os três passos na seguinte ordem.

0. se árvore vazia; fim
1. visitar nó raiz;
2. percorrer a sub-árvore da esquerda em pré- ordem;
3. percorrer a sub-árvore da direita em pré-ordem;

Para a árvore da figura 3.20 este percurso forneceria, no caso da visita significar imprimir, os seguintes resultados: **F B A D C E H G I**. Uma aplicação interessante deste tipo de percurso é aplicá-lo à uma árvore que contenha uma expressão aritmética, a qual foi expandida e recebeu todos os parênteses.

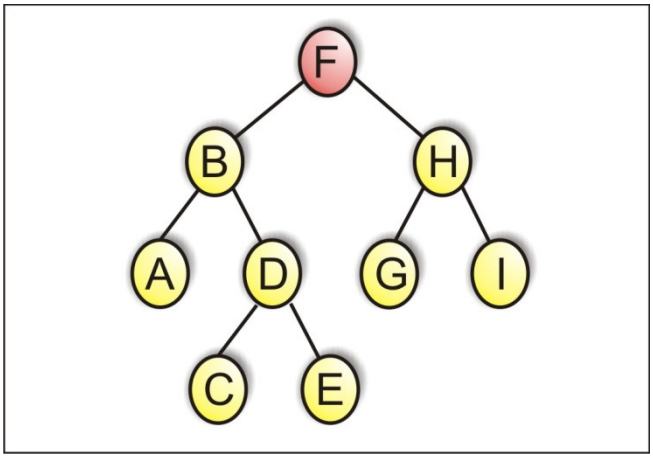


Figura 3.20: Percurso Pré-Ordem I.

A figura 3.21 mostra outro exemplo de árvore na qual um percurso é realizado em pré-ordem e obteve a seguinte seqüência de vértices visitados: **A B D C E G F H I.**

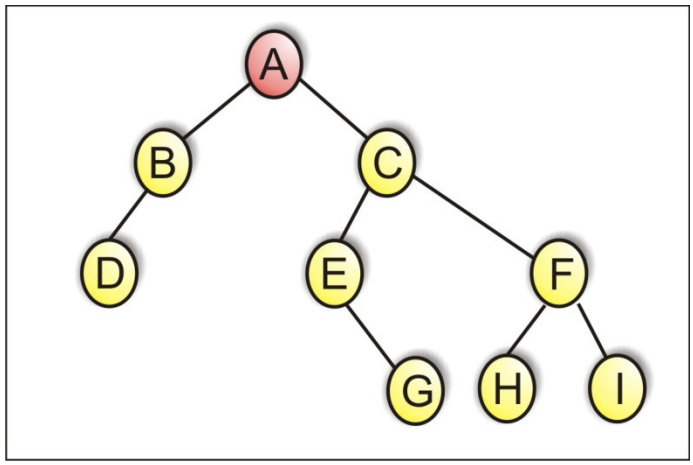


Figura 3.21: Percurso Pré-Ordem II.

Um algoritmo recursivo para implementar este modo de percurso em pré-ordem pode ser o seguinte:

```
proc pré-ordem (a:árvore)
        se a ≠ nil
        então
           início
                 execute visita(a);
                 execute pré-ordem(esq(a));
                 execute pré-ordem(dir(a))
           fim
```

Percurso In-Ordem ou em Ordem Simétrica: Para percorrer a árvore em ordem simétrica executa-se recursivamente os três passos na seguinte ordem:

0. se árvore vazia; fim
1. Percorrer a sub-árvore da esquerda in-ordem;
2. Visitar a raiz;
3. Percorrer a sub-árvore da direita in-ordem.

Considerando a árvore da figura 3.22, o percurso forneceria o seguinte resultado: **A B C D E F G H I.** Este tipo de percurso é muito empregado em árvores binárias de pesquisa.

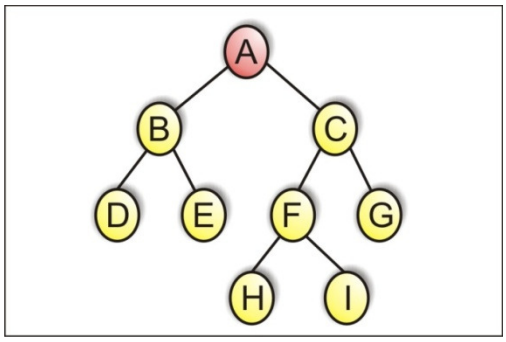


Figura 3.22: Percurso In-Ordem I.

A figura 3.23 mostra outro exemplo de árvore na qual um percurso é realizado in-ordem e obteve a seguinte seqüência de vértices visitados: **D B A E G C H F I.**

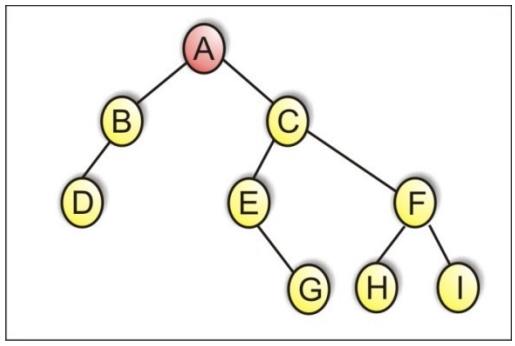


Figura 3.23: Percurso In-Ordem II.

Um algoritmo recursivo para implementar este modo de percurso in-ordem pode ser o seguinte:

```
proc in-ordem (a:árvore)
        se a ≠ nil
        então
           início
                 execute in-ordem(esq(a));
                 execute visita(a);
                 execute in-ordem(dir(a))
           fim
```

Percurso Pós-Ordem: O percurso conhecido como pós-ordem é feito a partir dos três passos na seguinte ordem:

0. se árvore vazia; fim
1. percorrer a sub-árvore da esquerda em pós-ordem;
2. percorrer a sub-árvore da direita em pós-ordem;
3. visitar nó raiz;

Para a árvore da figura 3.24 o percurso forneceria o seguinte resultado **A C E D B G I H F**. O percurso em pós-ordem pode ser aplicado no cálculo da altura de uma árvore.

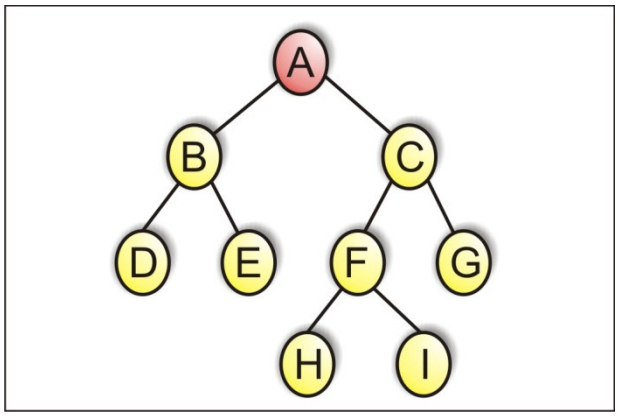


Figura 3.24: Percurso Pós-Ordem I.

A figura 3.25 mostra outro exemplo de árvore na qual um percurso é realizado pós-ordem e obteve a seguinte seqüência de vértices visitados: **D B G E H I F C A.**

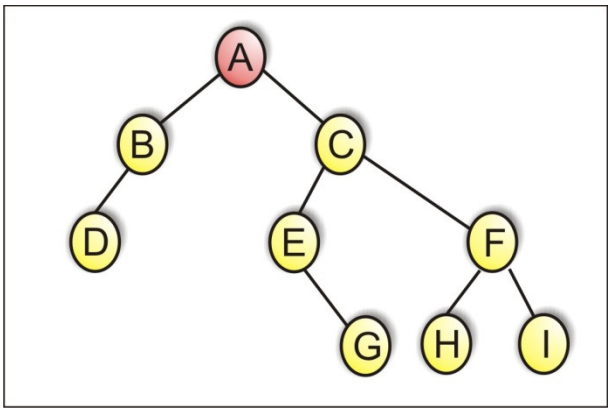


Figura 3.25: Percurso Pós-Ordem II.

Um algoritmo recursivo para implementar este modo de percurso em pré-ordem pode ser o seguinte:

```
proc pós-ordem (a:árvore)
        se a ≠ nil
        então
           início
                execute pós-ordem(esq(a));
                execute pós-ordem(dir(a))
                execute visita(a);

           fim
```

Praticar: Para a árvore da figura 3.20 qual o conjunto de vértices visitados em um percurso in-ordem? E para a figura 3.24 qual o conjunto de vértices visitados em um percurso realizado em pré-ordem?

2.4. Altura e número de nós

Se cada nó é capaz de ser ter dois filhos, então em um determinado nível, é possível ter no máximo *2n* nós nesse nível, onde *n* é o número de nós máximo do nível anterior. Então o número de nós máximo de uma árvore de uma determinada altura (nível), é igual a *2n+n*. Logo podemos esboçar os seguintes algoritmos recorrentes para calcular o número máximo de nós por nível (*max-nós-nível*) e o número máximo de nós da árvore (*max-nós-árvore*):

```
proc max-nós-nível (n:int)
        se n ≠ 1
        então
           início
                retorne(2*nós-nível(n-1))
           fim
```

```
proc max-nós-árvore (n:int)
        n_max_arvore:int;
        se n ≠ 1
        então
           início
                n_max_arvore = max-nós-árvore(n-1);
                retorne(2* n_max_arvore + n_max_arvore)
           fim
```

2.5. Conversão em árvore binária

Uma árvore qualquer pode ser transformada em uma árvore binária se ligarmos os irmãos e removermos a ligação entre um nó pai e os nós filhos, exceto os do primeiro filho.

Seja *T* uma árvore qualquer. *T* é convertida em uma árvore binária *B(T)* da seguinte maneira:

· *B(T)* possui um nó *B(v)* para cada nó *v* de *T*;

· As raízes de *T* e *B(T)* coincidem;

· O filho esquerdo de um nó *B(v)* em *B(T)* corresponde ao primeiro filho de *v* em *T*, caso exista. Se não existir, a sub-árvore esquerda de *B(v)* é vazia;

· O filho direito de um nó *B(v)* em *B(T)* corresponde ao irmão de *v* em *T*, localizado imediatamente a sua direita, caso exista. Se não existir, a sub-árvore direita de *B(v)* é vazia.

A figura 3.26 mostra uma árvore genérica convertida para uma árvore binária.

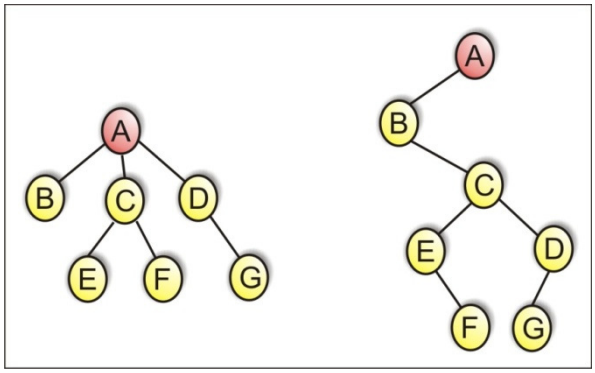
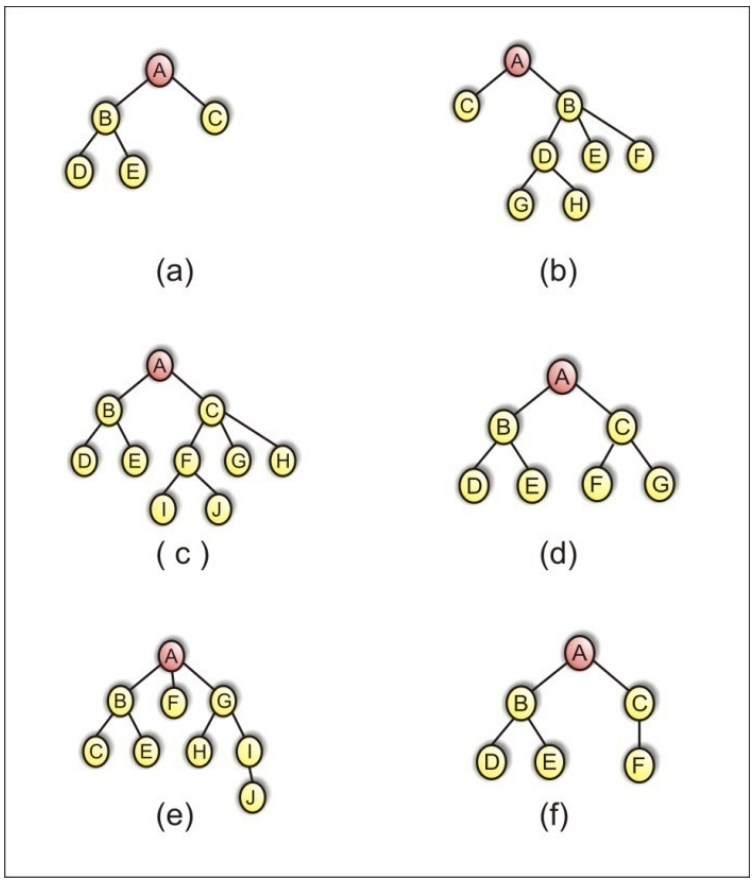


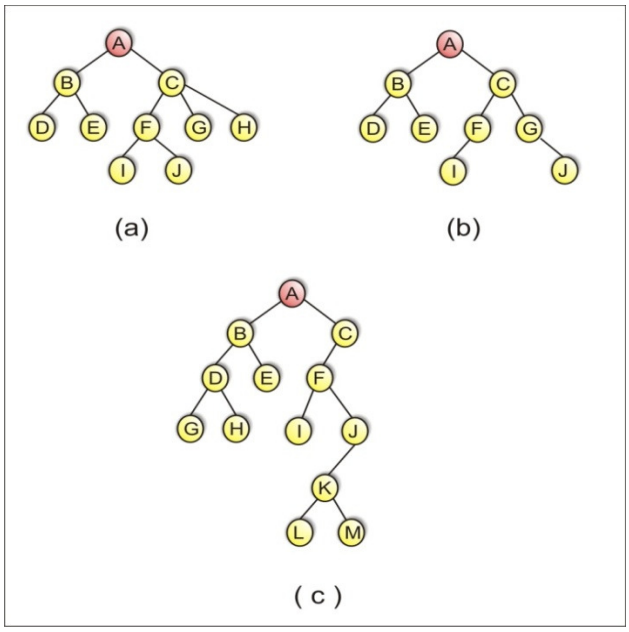
Figura 3.26: Conversão em árvore binária.

2.6. Exercícios

1. Quais das árvores abaixo são binárias? Justifique sua resposta utilizando a teoria apresentada.



2. Considerando as árvores abaixo responda:



a. Qual a seqüência de vértices visitados em um percurso realizado em pré-ordem?

b. Qual a seqüência de vértices visitados em um percurso realizado in-ordem?

c. Qual a seqüência de vértices visitados em um percurso realizado em pós-ordem?

3. Converta as árvore não binárias da questão 1 em árvores binárias.

4. Encontre o número máximo de nós por nível e o número máximo de nós da árvore para as árvores apresentadas na questão 2 .

5. Implemente em Java os algoritmos para percurso em pré-ordem, in-ordem e pós-ordem. Verifique a corretude dos mesmos comparando as suas respostas da questão 2 com a saída do programa.

6. Implemente em Java os algoritmos para calcular o número máximo de nós por nível e o número máximo de nós da árvore. Verifique a corretude dos mesmos comparando as suas respostas da questão 4 com a saída do programa.

UNIDADE II
ÁRVORES

3. ÁRVORES DE PESQUISA

3.1. Introdução

A operação de busca é encontrada com muita freqüência em aplicações computacionais, sendo, portanto importante estudar estratégias distintas para efetuá-la. Por exemplo, um programa de controle de estoque pode buscar, dado um código numérico ou um nome, a descrição e as características de um determinado produto. Se, temos um grande número de produtos cadastrados, o método para efetuar a busca deve ser eficiente, caso contrário a busca pode ser muito demorada, inviabilizando sua utilização.

Conforme veremos, certos métodos de organizar dados tornam o processo de busca mais eficiente. Como a operação de busca é uma tarefa muito comum em computação, o conhecimento desses métodos é de grande importância para a formação de um bom programador.

Antes de explicarmos os métodos, vamos definir alguns termos. Uma tabela ou arquivo é um grupo de elementos, cada uma dos quais chamados de registro. Existe uma chave associada a cada registro, usada para diferenciar os registros entre si.

Para todo arquivo, existe pelo menos um conjunto exclusivo de chaves (possivelmente mais). Este tipo de chave é chamado de chave primária. Entretanto, como todo campo de um registro pode servir como chave em uma determinada aplicação, nem sempre as chaves precisam ser exclusivas. Esse tipo de chave é chamado de chave secundária.

Quanto ao modo de organização da tabela ou do arquivo, ele pode ser um vetor de registros, uma lista encadeada, uma árvore, etc. Como diferentes técnicas de busca podem ser adequadas a diferentes organizações de tabelas, uma tabela é freqüentemente elaborada com uma técnica de busca em mente.

A tabela pode ficar totalmente contida na memória interna, totalmente na memória externa, ou pode ser dividida entre ambas. É evidente que são necessárias diferentes técnicas de pesquisa sob essas diferentes premissas.

Estudaremos agora, algumas estratégias de busca. Inicialmente, consideraremos que temos nossos dados armazenados em um vetor e discutiremos os algoritmos de busca que podemos empregar. A seguir, discutiremos a utilização de árvores de pesquisa, que são estruturas de árvores projetadas para darem suporte a operações de busca de forma eficiente.

Pesquisa Seqüencial

Este é o método mais simples de pesquisa. Consiste em uma varredura serial da tabela, durante a qual o argumento de pesquisa é comparado com a chave de cada registro até ser encontrada uma que seja igual, ou ser atingido o final da tabela, caso a chave procurada não se encontre na tabela.

A seguir é apresentado o algoritmo de pesquisa seqüencial em uma tabela não ordenada.

Adotaremos para a nossa implementação um método que recebe como parâmetro um vetor de elementos inteiros (v), a quantidade de elementos do vetor (n) e a chave a ser pesquisada (k), retornando o endereço (índice) do elemento encontrado, ou –1 caso contrário.

```
public static int buscaSequencial( int[ ] v, int n, int k)
{
  int i;
  for (i=0; i<n; i++)
     if (k == v[i])
        return i;                  // elemento encontrado
  return -1;                       // element não encontrado
}
```

Esse algoritmo de busca é extremamente simples, mas pode ser muito ineficiente quando o número de elementos no vetor for muito grande. Isto porque o algoritmo (a função, no caso) pode ter que percorrer todos os elementos do vetor para verificar que um determinado elemento está ou não presente. Dizemos que no pior caso será necessário realizar *n* comparações, onde *n* representa o número de elementos no vetor.

Portanto, o desempenho computacional desse algoritmo varia linearmente com relação ao tamanho do problema – chamamos esse algoritmo de busca linear.

Em geral, usamos a notação "Big-O" para expressarmos como a complexidade de um algoritmo varia com o tamanho do problema. Assim, nesse caso em que o tempo computacional varia linearmente com o tamanho do problema, dizemos que trata-se de um algoritmo de ordem linear e expressamos isto escrevendo *O(n)*.

No melhor caso, se dermos sorte do elemento procurado ocupar a primeira posição do vetor, o algoritmo acima necessitaria de apenas uma única comparação. Esse fato, no entanto, não pode ser usado para fazermos uma análise de desempenho do algoritmo, pois o melhor caso representa um situação muito particular.

Além do pior caso, devemos analisar o caso médio, isto é, o caso que ocorre na média. Já vimos que o algoritmo em questão requer *n* comparações quando o elemento não está presente no

vetor. E no caso do elemento estar presente, quantas operações de comparação são, em média, necessárias? Na média, podemos concluir que são necessárias *n/2* comparações. Em termos de ordem de complexidade, no entanto, continuamos a ter uma variação linear, isto é, *O(n)*, pois dizemos que *O(k n)*, onde *k* é uma constante, é igual a *O(n)*.

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

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

```
public static int buscaSequencialOrdenada( int[ ] vet, int n, int elem)
{
   int i;
   for (i=0; i<n; i++){
      if (elem == v[i])
         return i;                  // element encontrado
     else
        if (elem < vet[i])
         return -1;                 // interrompe busca
   }
   return -1;                       // percorreu todo o vetor e não encontrou o
elemento
}
```

No caso do elemento procurado não pertencer ao vetor, esse segundo algoritmo apresenta um desempenho ligeiramente superior ao primeiro, mas a ordem dessa versão do algoritmo continua sendo linear – *O(n)*. No entanto, se os elementos do vetor estão ordenados, existe um algoritmo muito mais eficiente que será apresentado a seguir.

Pesquisa Binária:

Caso os elementos do vetor estejam em ordem, podemos aplicar um algoritmo mais eficiente para realizarmos a busca. Trata-se do algoritmo de busca binária.

A idéia do algoritmo é testar o elemento que buscamos com o valor do elemento armazenado no meio do vetor. Se o elemento que buscamos for menor que o elemento do meio, sabemos que, se o elemento estiver presente no vetor, ele estará na primeira parte do vetor; se for maior, estará na segunda parte do vetor; se for igual, achamos o elemento no vetor. Se concluirmos que o elemento está numa das partes do vetor, repetimos o procedimento considerando apenas a parte que restou: comparamos o elemento que buscamos com o elemento armazenado no meio dessa parte. Este procedimento é continuamente repetido, subdividindo a parte de interesse, até encontrarmos o elemento ou chegarmos a uma parte do vetor com tamanho zero.

Em outras palavras, O método consiste na comparação do argumento de pesquisa (k) com a chave localizada no meio da tabela. Se for igual, a pesquisa termina com sucesso. Se (k) for maior, o processo é repetido para a metade superior da tabela e se for menor, para a metade inferior.

Para exemplificar, suponha que você tenha um conjunto de elementos a armazenar no seu computador, de forma que possa realizar inúmeras buscas sobre esses elementos. Como vocês devem lembrar, a complexidade de uma busca linear é O(n), onde n é o número de elementos da lista. Isso ocorre porque precisamos percorrer a lista, no pior caso, até o final. E se pudéssemos dar "saltos" nessa procura?

Claro que podemos dar "saltos" em uma procura. Aliás, essa é uma forma natural de otimizar processos. Para melhor visualizar esse processo, imagine que a busca, ao invés de utilizar uma lista como estrutura, possui um vetor de tamanho fixo, com elementos dispostos em ordem crescente e possuindo por exemplo, 100 elementos. Ao invés de iniciar sua busca no elemento inicial (0) e se entender, você pode testar o médio (50) e verificar se o número a ser encontrado é maior ou menor que o valor contido naquele índice. Se for menor, como o vetor está em ordem crescente, podemos desprezar todos os números acima de 50 e recomeçar a busca no intervalo de 0 a 50. Esse procedimento deverá se repetir até que o número correspondente seja encontrado ou não exista no vetor.

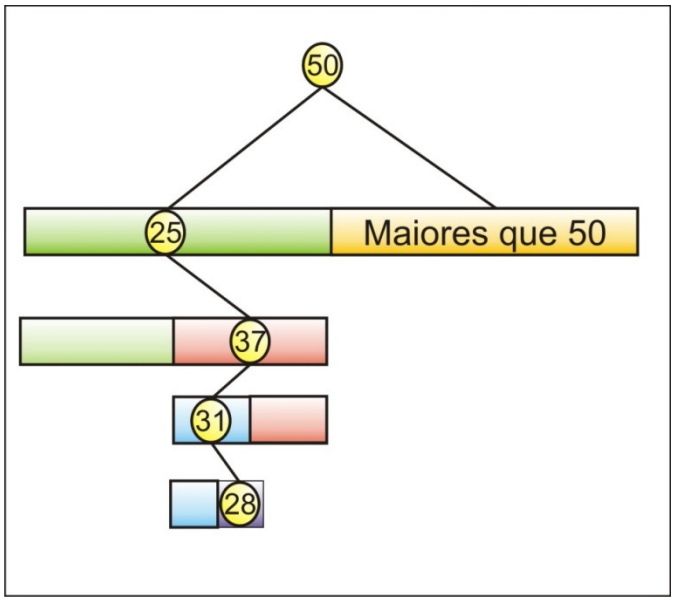


Figura 3.27: Pesquisa binária.

O exemplo da figura 3.27 ilustra o processo de busca de um elemento na posição (28) do vetor. O processo começa a pesquisar no elemento médio (50) e, verificando que o elemento procurado é menor que o elemento da posição (50), todos os elementos da posição depois de (50) serão desprezados. Em seguida, o mesmo procedimento é realizado para o subconjunto 0 a 50, depois no subconjunto 25 a 50, etc., reduzindo, a cada iteração, metade dos possíveis elementos até chegar no elemento desejado, que, no exemplo é o contido na posição 28.

O código a seguir ilustra uma implementação de busca binária, de forma não recursiva, em um vetor de valores inteiros ordenados de forma crescente.

```
public static int buscaBinaria( int[ ] vet, int n, int elem)
{
// no início consideramos todo o vetor
   int i =0;
   int fim = n-1;
   int meio;

// enquanto a parte maior for maior que zero
   while (ini <= fim){
      meio = (ini + fim) / 2;
      if (elem < vet[meio])
         fim = meio -1;                    // ajusta posição final
     else if (elem > vet[meio])
         ini = meio + 1;                   // ajusta posição inicial
     else
        return meio;
   }
   retun -1;                               // elemento não encontrado
}
```

O desempenho desse algoritmo é muito superior ao de busca linear. Novamente, o pior caso caracteriza-se pela situação do elemento que buscamos não estar no vetor. Quantas vezes precisamos repetir o procedimento de subdivisão para concluirmos que o elemento não está presente no vetor? A cada repetição, a

parte considerada na busca é dividida à metade. A tabela abaixo mostra o tamanho do vetor a cada repetição do laço do algoritmo.

Repetição	*Tamanho do problema*
1	*n*
2	*n/2*
3	*n/4*
...	*...*
Log n	*1*

Sendo assim necessárias *log n* repetições. Como fazemos um número constante de comparações a cada ciclo (duas comparações por ciclo), podemos concluir que a ordem desse algoritmo é *O(log n)*.

O algoritmo de busca binária consiste em repetirmos o mesmo procedimento recursivamente, podendo ser naturalmente implementado de forma recursiva. Embora a implementação não recursiva seja mais eficiente e mais adequada para esse algoritmo, a implementação recursiva é mais sucinta e vale a pena ser apresentada.

Na implementação recursiva, temos dois casos a serem tratados. No primeiro, a busca deve continuar na primeira metade do vetor, logo chamamos a função recursivamente passando como parâmetros o número de elementos dessa primeira parte restante e o mesmo ponteiro para o primeiro elemento, pois a primeira parte tem o mesmo primeiro elemento do que o vetor como um todo. No segundo caso, a busca deve continuar apenas na segunda parte do vetor, logo passamos na chamada recursiva, além do número de elementos restantes, um ponteiro para o primeiro elemento dessa segunda parte. Para simplificar, a função de busca apenas informa se o elemento pertence ou não ao vetor, tendo como valor de retorno falso (0) ou verdadeiro (1). Uma possível implementação usando essa estratégia é mostrada a seguir.

```
public static int buscaBinariaRecursiva( int[ ] vet, int n, int elem)
{
  // testa a condição de contorno: parte com tamanho zero
  if (n<=0)
    retun 0;
  else{
  // deve buscar o elemento entre os índices 0 e n-1
    int meio = (n-1) / 2;
    if (elem < vet[meio])
      return buscaBinariaRecursica(vet,meio,elem);
    else if (elem > vet[meio])
      return buscaBinariaRecursica(vet[meio+1],n-1-meio,elem);
    else
      // elemento encontrado
      return 1;
  }
}
```

Praticar: Considere o vetor ordenado v1 = [12,25,43,59,68,87] e o vetor desordenado v2 = [25,59,43,68,12,87] cada um deles contendo seis elementos. Execute manualmente os algoritmos apresentados a fim de encontrar o elemento 43.

3.2. Definição

Observando com atenção, vamos notar que o processo de pesquisa ilustrado na figura 3.27 gerou implicitamente uma árvore. E se fizéssemos o contrário? Se ao invés da busca gerar uma árvore, termos uma árvore para fazer a busca? Neste caso, ao invés do processo de busca criar uma árvore, podemos definir uma árvore que facilite o processo de busca.

A vantagem dessa forma de representação é o custo associado ao processo de inserção e remoção de elementos no conjunto. Para melhor justificar essa proposição, suponha que temos um problema que precise fazer uma busca de elementos em um conjunto. Temos o conceito de busca binária e podemos utilizar um vetor, como descrito anteriormente, para facilitar a busca. Porém, se

no nosso problema, precisássemos repetidamente inserir novos elementos ao conjunto e remover elementos existentes o custo do pior caso dessas operações seria de O(n) tanto na inserção quanto na remoção, uma vez que teríamos que deslocar todos os elementos que se encontram à direita do novo ou existente elemento para a esquerda (remoção) ou para direita (inserção).

Para melhorar esse custo, podemos utilizar estrutura mais dinâmica que o vetor. Podemos utilizar uma árvore onde todos os seus elementos à esquerda de um determinado nó são sempre menores que o elemento do nó, e todos os elementos à direita são maiores. Esse tipo de árvore é conhecida como árvore de pesquisa.

Uma árvore de pesquisa é uma árvore estruturada de forma a facilitar a busca em seus elementos. Cada elemento deve estar associado a uma chave, cujo conjunto define a seqüência ordenada de busca dos elementos. Cada nó da árvore possui, portanto, uma chave e um elemento, e deve fazer com que todos os elementos cuja chave sejam menores que a sua se encontrem à esquerda e todos aqueles cuja chave sejam maiores estejam a sua direita, como o ilustrado na figura 3.28.

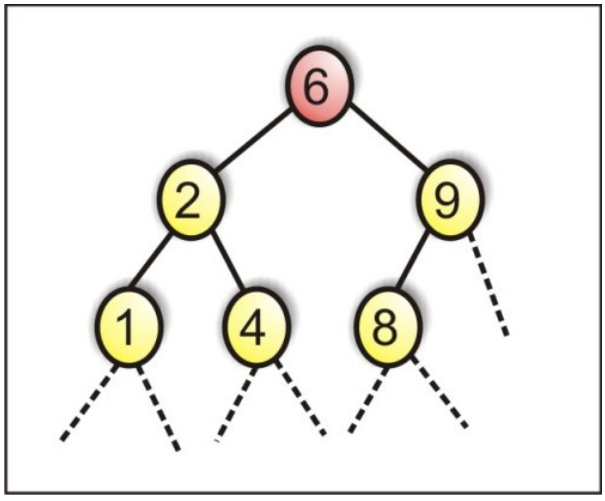


Figura 3.28: Árvore de Pesquisa.

É importante salientar que uma árvore de pesquisa não é necessariamente uma árvore binária, por exemplo, árvores de pesquisa com mais de uma chave por nó, como o ilustrado na figura 3.29. O importante em uma árvore de pesquisa é seguir a propriedade citada anteriormente.

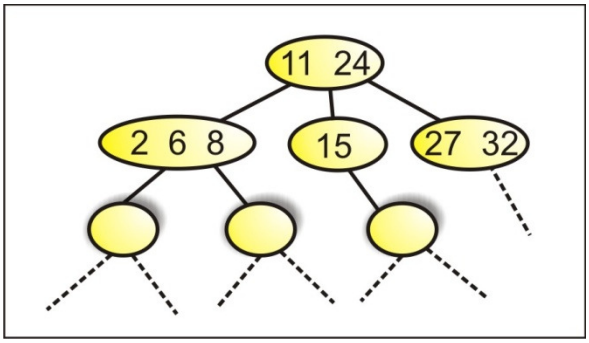
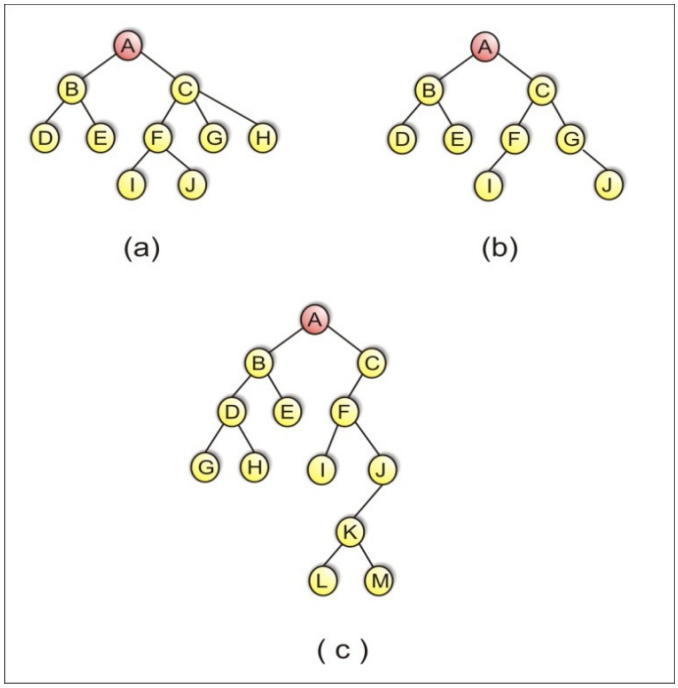


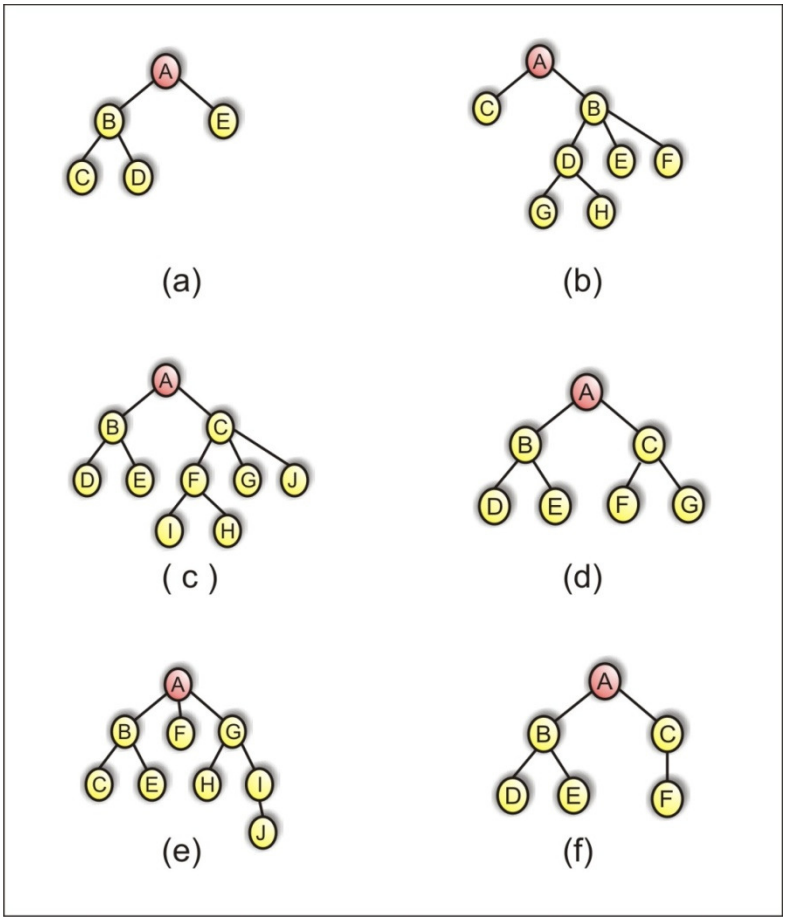
Figura 3.29: Árvore de Pesquisa não binária.

3.3. Exercícios

1. As árvores abaixo são árvores de pesquisa?



2. Quais das árvores abaixo são árvores de pesquisa? Justifique sua resposta utilizando a teoria apresentada.



3. Considere um vetor que possui no máximo quatrocentos elementos podendo estes estarem ordenados ou não. Implemente, em Java, um programa capaz de procurar um elemento neste vetor utilizando pesquisa seqüencial. Reimplemente o programa anterior agora utilizando pesquisa binária.

UNIDADE II
ÁRVORES

4. ÁRVORES BINÁRIA DE PESQUISA

4.1. Introdução

Árvores binárias de pesquisa possuem a propriedade das árvores de pesquisa apresentadas anteriormente com a restrição que cada nó possua no máximo dois filhos, um à esquerda e outro à direita, como ilustra a figura 3.30.

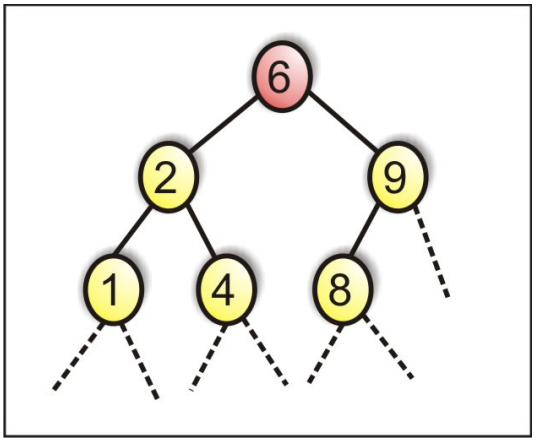


Figura 3.30: Árvore Binária de Pesquisa.

Formalizando, uma árvore binária de pesquisa T (ABP) ou árvore binária de busca é tal que T = ϕ e a árvore é dita vazia ou seu nó raiz contém uma chave é:

· Todas as chaves da sub-árvore esquerda são menores que a chave da raiz;
· Todas as chaves da sub-árvore direita são maiores que a chave raiz;
· As sub-árvores direita e esquerda são também árvores binárias de busca.

Uma possível implementação da estrutura de dados para uma árvore binária de pesquisa é:

```
class No
{
  int   iDado;           // dado usado como valor chave
  double   fDado;        // outro dado
  No   filhoDireita;     // filho à esquerda
  No   filhoEsquerda; // filho à direita
}
```

4.2. Operações

Várias operações podem ser realizadas sobre as estruturas árvores binárias de pesquisa, sendo as principais as operações de busca, inserção e remoção.

Para exemplificar a implementação de operações em árvores binárias de busca, vamos considerar o caso em que a informação associada a um nó é um número inteiro, e não vamos considerar a possibilidade de repetição de valores associados aos nós da árvore. A figura 3.31 ilustra uma árvore de busca de valores inteiros.

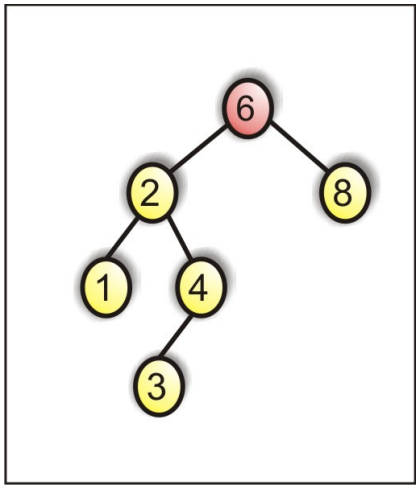


Figura 3.31: Árvore de busca com valores inteiros.

Considere a seguinte classe que apresenta como métodos as três principais operações realizadas em árvores: localizar, inserir e deletar.

```
class Arvore
{
  private  No raiz;                    // o único campo de dado em Arvore
  public  void  localizar(int chave){ }// localizar um elemento em Arvore
  public  void  inserir(int id, double dd ){ }// inserir um elemento em Arvore
 public  void  deletar(int id){ }      // apagar um elemento em Arvore
}
```

Esses métodos são análogos aos existentes em árvores binárias comuns, pois não exploram a propriedade de ordenação das árvores de busca. A seguir detalharemos cada uma dessas operações.

Busca

A operação para buscar um elemento na árvore explora a propriedade de ordenação da árvore, tendo um desempenho computacional proporcional a sua altura (*O(log n)* para o caso de árvore balanceada). Uma implementação do método de busca pode ser dado por:

```
public No localizar(int chave)              // assume-se árvore não vazia
{
  No   noCorrente = raiz;                   // começa na raiz
  while (noCorrente.iDado != chave){
    if (chave < noCorrente.iDado)           // ir para esquerda?
      noCorrente = noCorrente.filhoEsquerda;
    else                                    // ou para direita?
      noCorrente = noCorrente.filhoDireita;
    if (noCorrente = null)                  // elemento não encontrado
      return null;
  }
  return noCorrente;                        // elemento encontrado
}
```

A figura 3.32 apresenta a busca pelo elemento 18 na árvore.

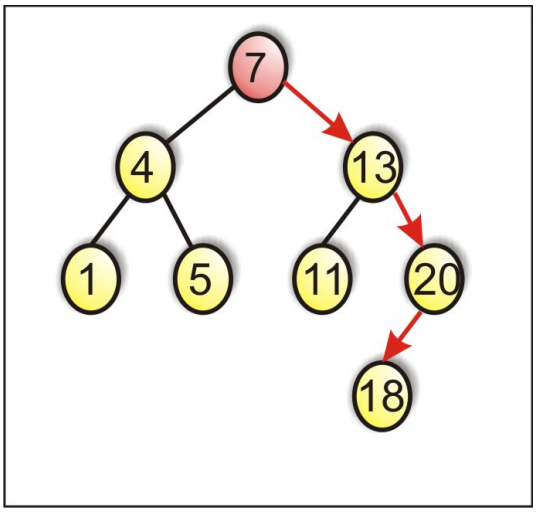


Figura 3.32: Localizar um elemento na árvore.

Praticar: Utilize a figura 3.32 e agora verifique como ela seria caso a busca fosse pelo elemento 5.

Inserção

A operação de inserção adiciona um elemento na árvore na posição correta para que a propriedade fundamental das árvores de pesquisa seja mantida. Para inserir um valor v em uma árvore usamos sua estrutura recursiva, e a ordenação especificada na propriedade fundamental. Se a (sub-) árvore é vazia, deve ser substituída por uma árvore cujo único nó (o nó raiz) contém o valor v. Se a árvore não é vazia, comparamos v com o valor na raiz da árvore, e inserimos v na sub-árvore esquerda ou na sub-árvore direita, conforme o resultado da comparação. É importante salientar que os novos elementos inseridos em uma árvore entram sempre na condição de folhas.

Em resumo o processo de inserção desenvolve-se da seguinte forma:

1. Se a árvore for vazia, o símbolo é instalado na raiz;
2. Caso contrário, é instalado na sub-árvore esquerda, se for menor que o símbolo raiz, ou da direita se for maior;
3. Se for igual ao símbolo raiz, o elemento não será inserido na árvore;

O método abaixo ilustra a implementação dessa operação.

```
public void inserir(int iDad, double fDad )          // assume-se árvore não vazia
{
  No newNo = new No();                               // cria novo nó
  newNo.iDado = iDad;                                // insere dado
  newNo.fDado = fDad;
  if (raiz == null)                                  // sem nó na raiz
    raiz = newNo;
  else
  {                                                  // raiz ocupada
    No noCorrente = raiz;                            // começa na raiz
    No progenitor;
    while(true)
      {
        progenitor  = noCorrente;
        if  (iDad < noCorrente.iDado)                // vai para esquerda?
          {
            noCorrente = noCorrente.filhoEsquerda;
            if (noCorrente = null)
              {
                progenitor.filhoEsquerda = newNo; // insere a esquerda
                return;
              }
          }                                          // fim do if ir para esquerda
        else                                         //ou para direita?
          {
            noCorrente = noCorrente.filhoDireita;
            if (noCorrente == null)
              {
                progenitor.filhoDireita = newNo;     // inserir a direita
                return;
              }
```

```
            }                                   // fim do else ir para
direita
        }                                       //fim do while
    }                                           // fim do else não raiz
}                                               // fim de inserir
```

Para entendermos o funcionamento do algoritmo de inserção, veja o exemplo de construção de uma árvore binária de pesquisa com o conjunto de números {17,99,13,1,3,100,400} apresentado na figura 3.33.

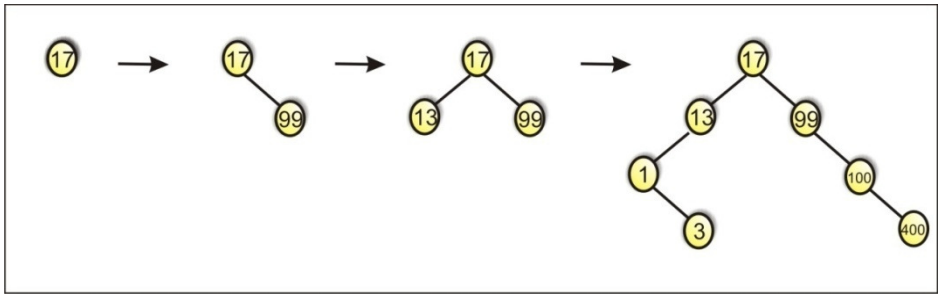


Figura 3.33: Construção de uma árvore.

Para iniciar é necessário a definição da raiz da árvore. No nosso caso o elemento (17) desempenha tal função. Em seguida inserimos o elemento (99), que é maior que (17) e, portanto é inserido na sub-árvore direita. Logo após, é inserido o elemento (13), menor que (17) sendo inserido na sub-árvore esquerda. O elemento (1), por sua vez é menor que a raiz (17) e o elemento (13) é então inserido na sub-árvore esquerda, sub-árvore essa que possui raiz o elemento (13). O elemento (3) é menor que a raiz (17) e maior que o elemento (1), sendo assim inserido na sub-árvore direita com raiz (1). O processo é repetido até que todos os elementos componham a árvore binária de pesquisa.

Praticar: Monte passo a passo uma árvore utilizando o algoritmo de inserção apresentado. Para tal, considere que a árvore binária de pesquisa é o conjunto de números {37,96,23,16,3,103,90}

Remoção

Outra operação a ser analisada é a que permite retirar um determinado elemento da árvore. Essa operação é um pouco mais complexa que a de inserção. Existem três situações possíveis:

1. Nó sem filhos: A primeira, e mais simples, é quando se deseja retirar um elemento que é folha da árvore (isto é, um elemento que não tem filhos). Neste caso, basta retirar o elemento da árvore e atualizar o pai, pois seu filho não existe mais. A figura 3.34 ilustra a remoção de uma folha.

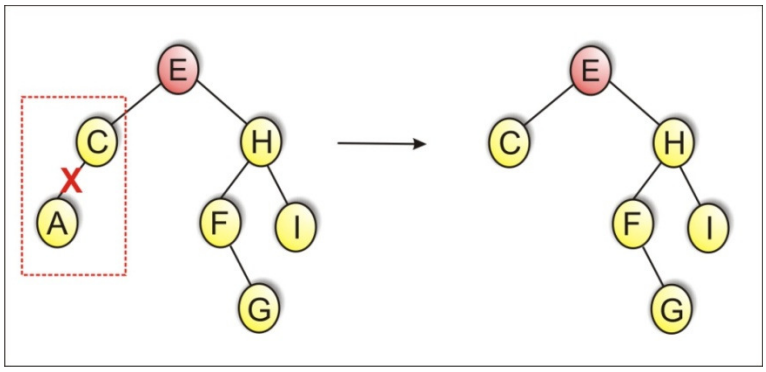


Figura 3.34: Remoção de um nó sem filhos.

2. Nó com um único filho: A segunda situação, ainda simples, acontece quando o nó a ser retirado possui um único filho. Para retirar esse elemento é necessário antes atualizar o pai, fazendo-o apontar para o neto. A figura 3.35 ilustra esse procedimento.

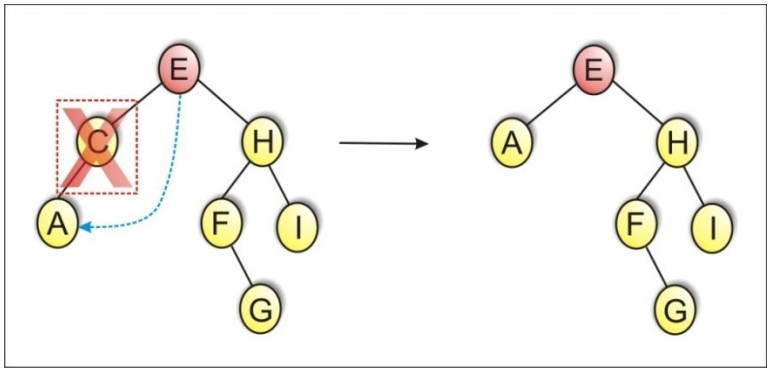


Figura 3.35: Remoção de um nó com um filho.

Um detalhamento sobre o processo de retirada de um nó que possui apenas um filho pode verificado na figura 3.36. Esta imagem foi retirada do material do Julio Cesar de Andrade Vieira Lopes (mais informações consultar as referências da unidade).

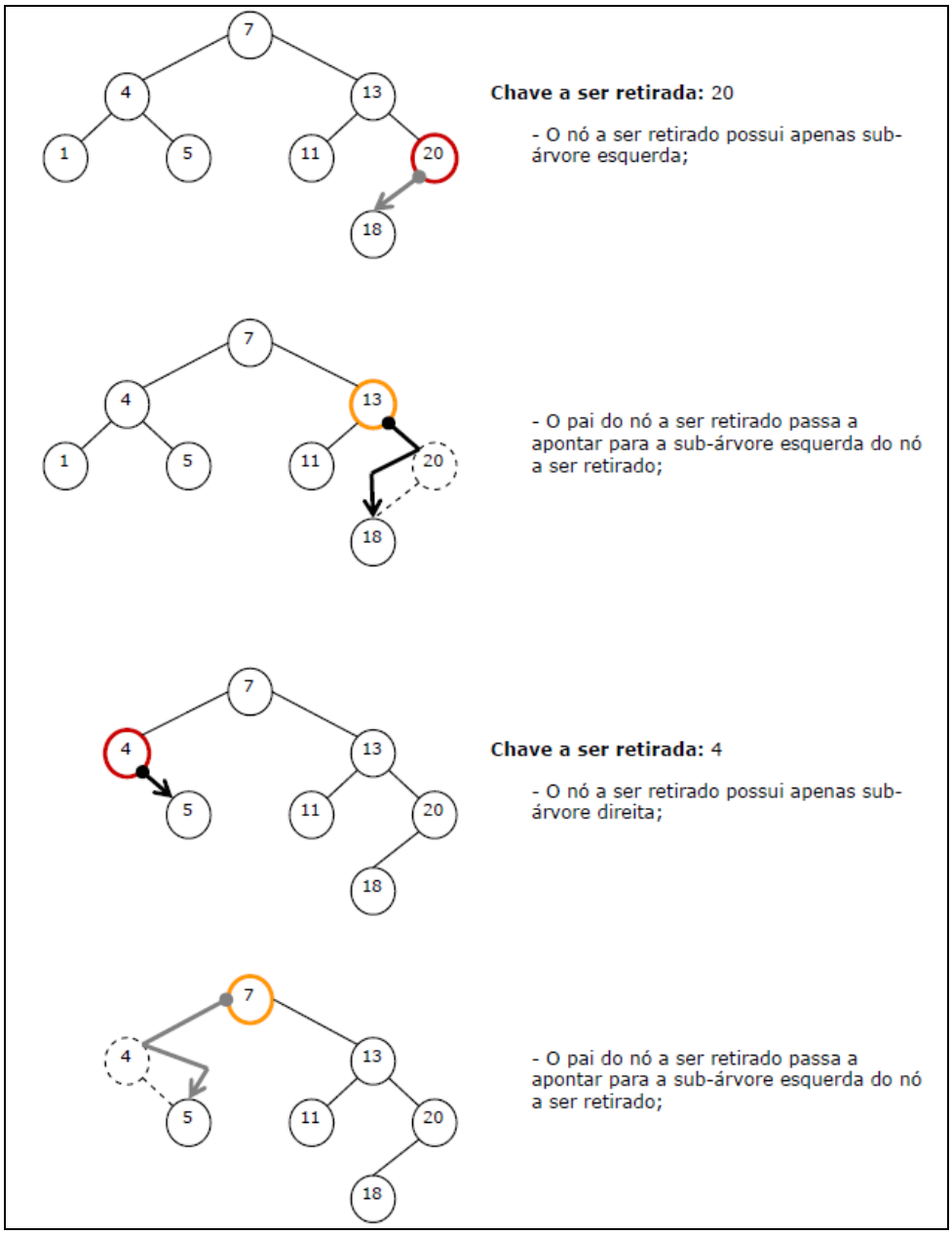


Figura 3.36:Detalhamento do processo de retirada de um nó com apenas um filho.

3. Nó com dois filhos: O caso complicado ocorre quando o nó a ser retirado tem dois filhos. Para poder retirar esse nó da árvore, devemos proceder da seguinte forma:

a) encontramos o elemento que precede o elemento a ser retirado na ordenação. Isto equivale a encontrar o elemento mais à direita da sub-árvore à esquerda;

b) trocamos a informação do nó a ser retirado com a informação do nó encontrado;

c) retiramos o nó encontrado (que agora contém a informação do nó que se deseja retirar). Observa-se que retirar o nó mais à direita é trivial, pois esse é um nó folha ou um nó com um único filho (no caso, o filho da direita nunca existe).

O procedimento descrito acima deve ser seguido para não haver violação da ordenação da árvore. Observamos que, análogo ao que foi feito com o nó mais à direita da sub-árvore à esquerda, pode ser feito com o nó mais à esquerda da sub-árvore à direita (que é o nó que segue o nó a ser retirado na ordenação). A figura #.8 ilustra o acima exposto.

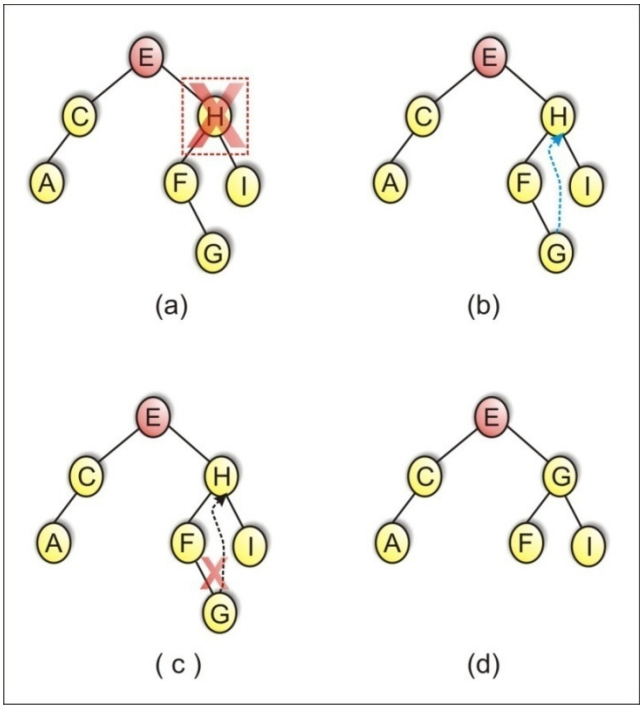


Figura 3.37: Remoção de um nó com dois filhos.

No caso da retirada de um nó com ambos os filhos, deve-se criar uma rotina aqui chamada de *pega_maior*(), que deverá pegar o maior elemento da sub-árvore que está sendo excluída.

Neste caso, como o *pega_maior*() deve ser aplicado sobre a sub-árvore esquerda do H, o valor a retornar pela função será o G.

Se, por exemplo, tivesse um F2 “pendurado” na esquerda do “G”, após a retirada do G ele deveria aparecer “pendurado” na direita do “F”, como mostra a figura 3.38.

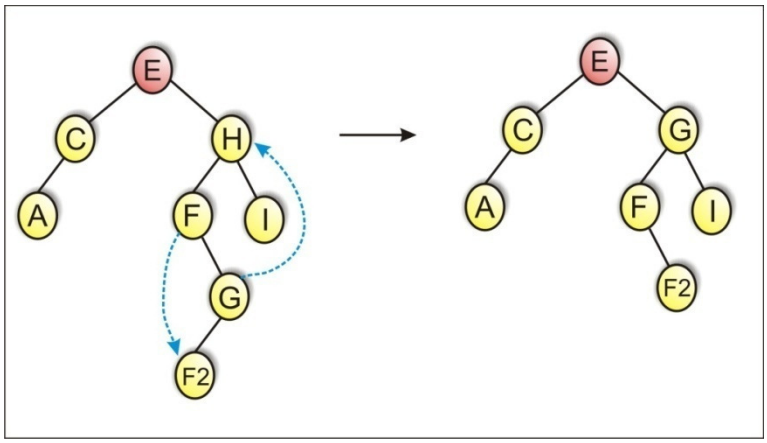


Figura 3.38: Remoção de um nó com dois filhos.

Praticar: Utilize a árvore binária de pesquisa montada no Praticar anterior e execute operações remoção de nós sem filhos, nós com um filho e nós com dois filhos.

As figuras 3.39 e 3.40 também retiradas do material de Julio Cesar de Andrade Vieira Lopes mostram com detalhes cada passo do processo de retirada de um nó com dois filhos.

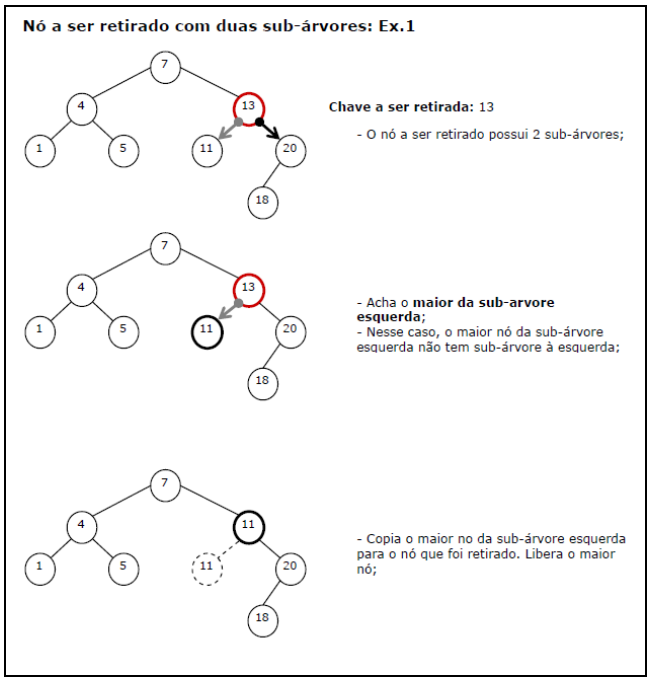


Figura 3.39:Remoção de um nó com dois filhos – Exemplo 01.

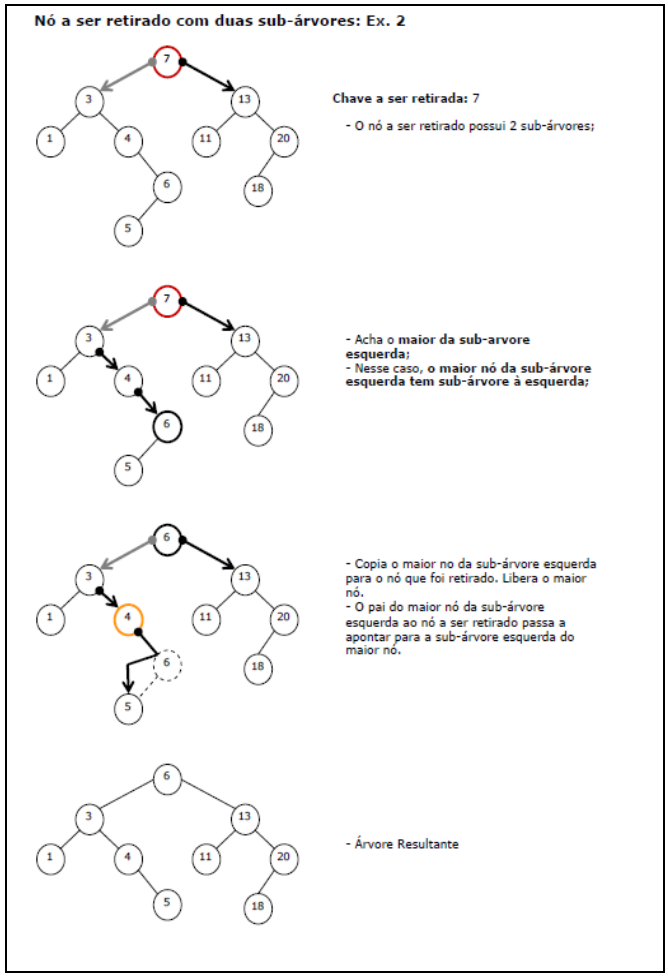
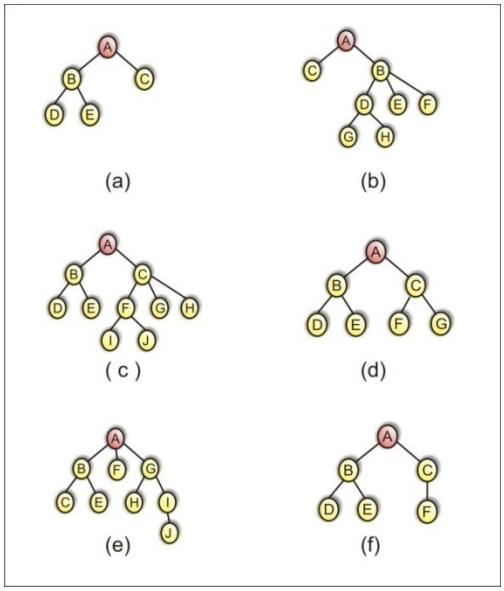


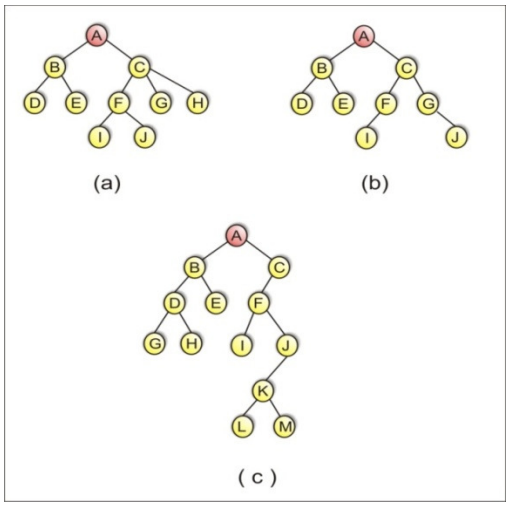
Figura 3.40:Remoção de um nó com dois filhos – Exemplo 02.

4.3. Exercícios

1. Quais das árvores abaixo são árvores binárias de pesquisa? Justifique sua resposta utilizando a teoria apresentada.



2. Transforme as árvores abaixo em árvores binárias de pesquisa.



3. Implemente, em Java, um programa que realize as três operações básicas em árvores binárias: incluir, excluir e pesquisar.

UNIDADE II
ÁRVORES

5. ÁRVORES AVL

No capítulo anterior foram apresentadas as operações relacionadas às árvores binárias de pesquisa. Estas operações e o conceito de árvore balanceada serão necessários para o entendimento das árvores AVL.

5.1 Balanceamento

Uma árvore é dita balanceada quando, para qualquer nó, as suas sub-árvores à esquerda e à direita possuem a mesma altura. Na figura 3.41 (a) é apresenta uma árvore balanceada e em (b) uma árvore desbalanceada.

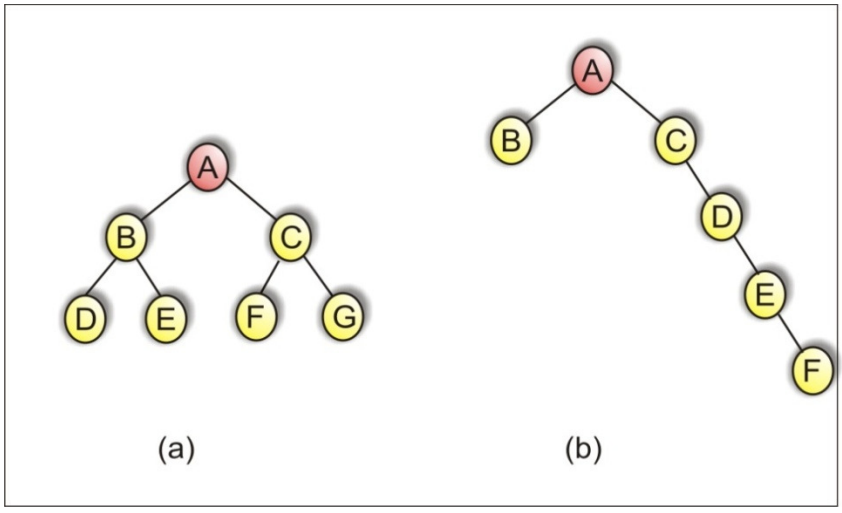


Figura 3.41: (a) Árvore balanceada (b) Árvore desbalanceada.

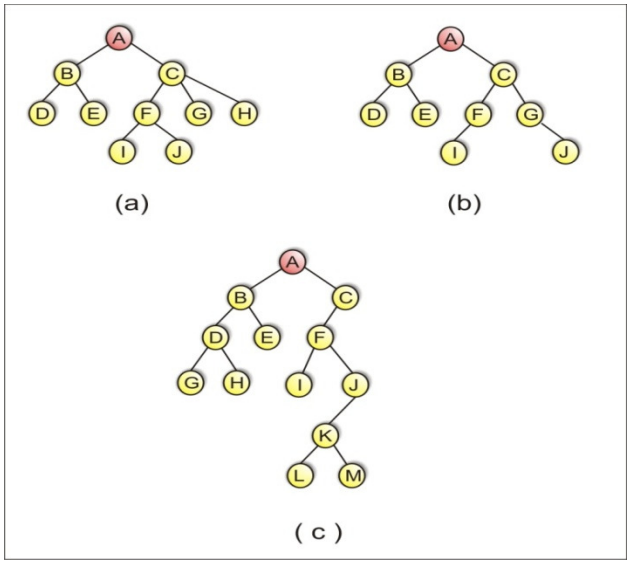
É fácil prever que, após várias operações de inserção/remoção, a árvore tende a ficar desbalanceada, já que essas operações, conforme descritas, não garantem o

balanceamento. Em especial, nota-se que a função de remoção favorece uma das sub-árvores (sempre retirando um nó da sub-árvore à esquerda, por exemplo). Uma estratégia que pode ser utilizada para amenizar o problema é intercalar de qual sub-árvore será inserido ou retirado o nó. No entanto, isso ainda não garante o balanceamento da árvore.

Para que seja possível usar árvores binárias de pesquisa mantendo sempre a altura das árvores no mínimo, ou próximo dele, é necessário um processo de inserção e remoção de nós mais complicado, que mantenha as árvores "balanceadas", ou "equilibradas", tendo as duas sub-árvores de cada nó o mesmo "peso", ou pesos aproximadamente iguais. No caso de um número de nós par, podemos aceitar uma diferença de um nó entre a *sae* (sub-árvore à esquerda) e a *sad* (sub-árvore à direita).

A idéia central de um algoritmo para balancear (equilibrar) uma árvore binária de pesquisa pode ser a seguinte: se tivermos uma árvore com *m* elementos na *sae*, e *n* >= *m* + 2 elementos na *sad*, podemos tornar a árvore menos desequilibrada movendo o valor da raiz para a *sae*, onde ele se tornará o maior valor, e movendo o menor elemento da *sad* para a raiz. Dessa forma, a árvore continua com os mesmos elementos na mesma ordem. A situação em que a *sad* tem menos elementos que a *sae* é semelhante. Esse processo pode ser repetido até que a diferença entre os números de elementos das duas sub-árvores seja menor ou igual a *1*. Naturalmente, o processo deve continuar (recursivamente) com o balanceamento das duas sub-árvores de cada árvore. Um ponto a observar é que remoção do menor (ou maior) elemento de uma árvore é mais simples do que a remoção de um elemento qualquer.

Praticar: As árvores abaixo são balanceadas? Justifique sua resposta utilizando a teoria apresentada.



5.2 Definição

As árvores AVL foram criadas em 1962 por Adelson-Velskii e Landis e historicamente foram a primeira estrutura de dados a oferecer operações de inserção, remoção e busca em tempo logaritmo ou seja é um algoritmo muito rápido.

Uma árvore binária balanceada, chamada de árvore AVL, é uma árvore binária na qual as alturas das duas sub-árvores de cada um dos nós nunca diferem em mais de 1. O balanceamento de um nó é igual à diferença entre as suas alturas esquerda e direita. Portanto, cada nó de uma árvore balanceada tem balanceamento igual a -1, 0 ou 1, dependendo da comparação entre as alturas esquerda e direita. Lembrando que a altura de um nó *n* da árvore é o número de nós do maior caminho de *n* até um de seus descendentes. As folhas tem altura 1. A figura 3.42 apresenta árvores AVL onde os números dos nodos representam o conteúdo e o fator de balanceamento para cada nodo.

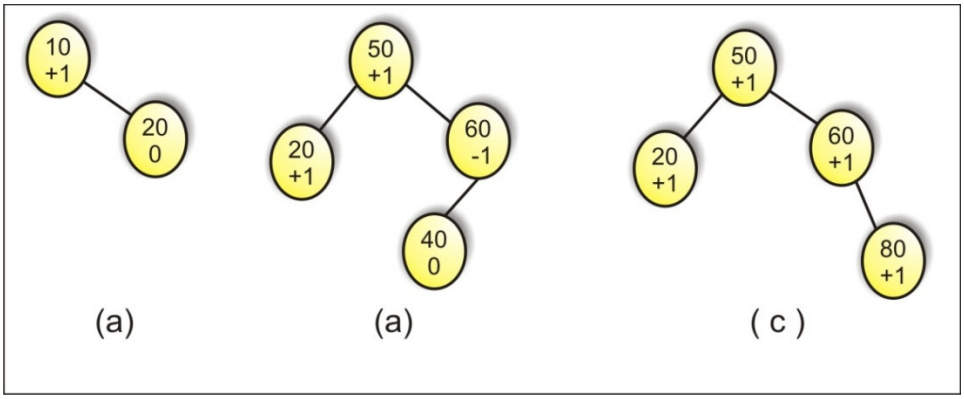


Figura 3.42: Árvore AVL com fator de balanceamento.

Para fortalecer o conceito de fator de balanceamento, considere outro exemplo, mostrado na figura 3.43, contendo uma árvore AVL e uma árvore NÃO-AVL.

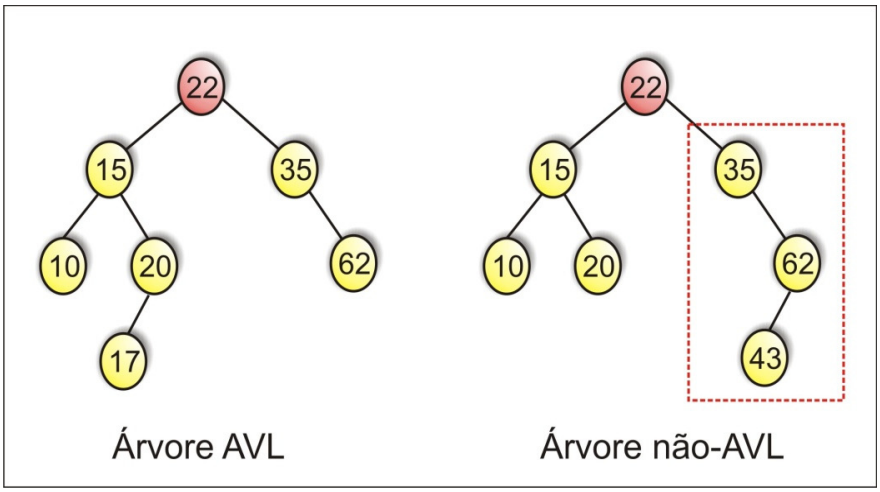


Figura 3.43: Árvore AVL e Árvore não AVL.

No exemplo da árvore não-AVL, o nodo 35 possui uma sub-árvore à direita com dois nodos (62 e 43) enquanto nenhuma à esquerda, ou seja, a altura entre as sub-árvores de 35 ultrapassou um nodo. Tal diferença chama-se “fator de balanceamento” (FB) e esta informação deverá constar em cada nó de uma árvore balanceada.

Assim, para cada nodo, pode-se definir um fator de balanceamento (FB) que vem a ser um número inteiro igual a:

$$FB(nodo) = altura(sad) - altura(sae)$$

sabendo-se que:

- o FB de uma folha é sempre zero (0);

- para uma árvore ser AVL, os FBs devem ser necessariamente -1, 0, ou 1: -1 para altura maior à esquerda; 0 se folha, +1 se a sub-árvore da direita possui altura maior.

Considerando a árvore AVL do exemplo anterior, os FBs estão indicados ao lado de cada nodo na figura 3.44.

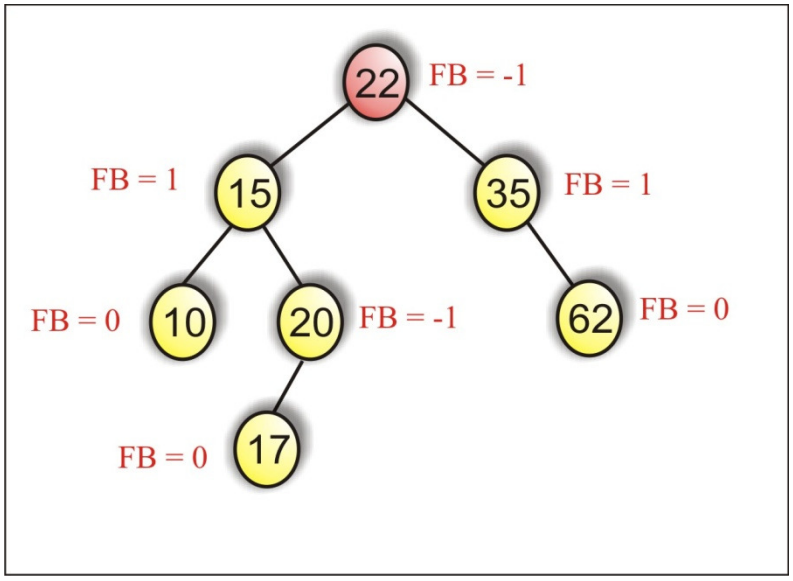


Figura 3.44: Fator de balanceamento.

5.3 Construção

A descrição acima pode ser resumida no seguinte algoritmo em pseudo-código para construção de uma árvore AVL:

1. Insira o novo nodo normalmente (ou seja, da mesma maneira que inserimos numa ABP);

2. Iniciando com o nodo pai do nodo recém-inserido, teste se a propriedade AVL é violada neste nodo, ou seja, teste se o FB deste nodo é maior do que **abs(1)**. Temos aqui 2 possibilidades:

2.1 A condição AVL foi violada

2.1.1 Execute as operações de rotação conforme for o caso (Tipo 1 ou Tipo 2)

2.1.2 Volte ao passo 1;

2.2 A condição AVL não foi violada. Teste pela condição AVL o pai do nodo testado por último (ou seja, retorne ao passo 2). Se o nodo recém-testado não tem pai, ou seja, é o nodo raiz da árvore, volte para inserir novo nodo (Passo 1)

O importante a observar neste algoritmo é que o teste por desbalanço inicia com o último nodo inserido, e não como nodo raiz.

Vamos ver um exemplo de construção de uma árvore AVL com os seguintes números: [10, 20, 30, 25, 27]. A inserção dos 3 primeiros números resulta na seguinte árvore:

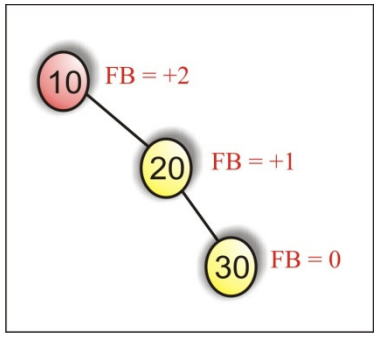


Figura 3.35: Montando uma árvore AVL 01.

Após a inserção do elemento 30 a árvore fica desbalanceada. O caso acima é do Tipo 2. Fazemos uma rotação para a esquerda no nodo com FB 2. A árvore resultante fica:

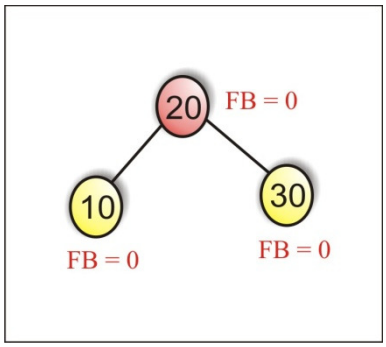


Figura 3.36: Montando uma árvore AVL 02.

O passo seguinte é inserir os nodos 25 e 27. A árvore fica desbalanceada apenas após a inserção do nodo 27, exemplificado abaixo:

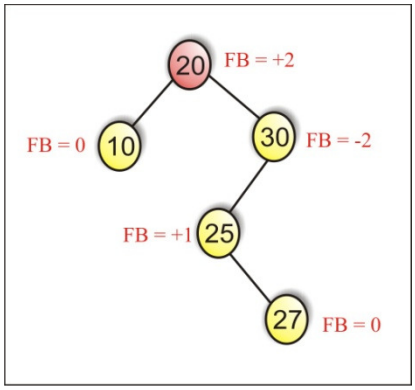


Figura 3.37: Montando uma árvore AVL 03.

Este caso é do Tipo 1. O nodo 30 tem FB -2 e o seu nodo filho tem FB 1. Precisamos efetuar uma rotação dupla, ou seja, uma rotação simples à esquerda do nodo 25, resultando:

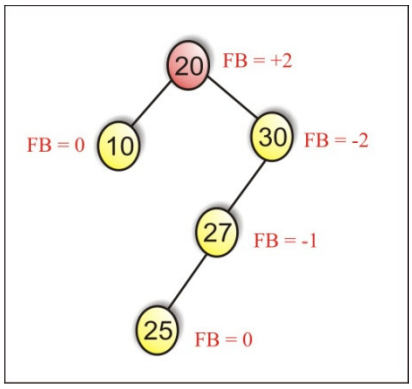


Figura 3.38: Montando uma árvore AVL 03.

Em seguida faz-se uma rotação simples à direita do nodo 30, resultando:

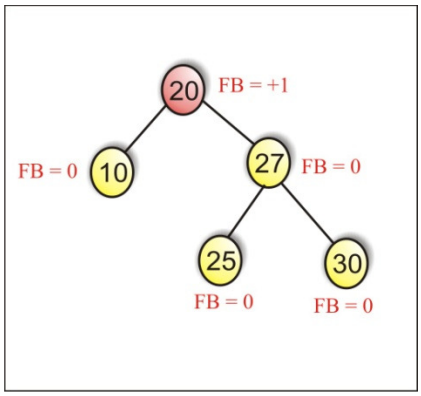


Figura 3.39: Montando uma árvore AVL 03.

e a árvore está balanceada.

Praticar: Construa uma árvore AVL com os seguintes números [13, 21, 36, 25, 29].

5.4 Operações

Apresentado de forma sucinta a metodologia de construção de uma árvore AVL, agora detalharemos as duas principais operações relacionadas a elas: inserção e remoção.

Inserção:

A inserção em uma árvore de AVL pode ser realizada introduzindo o valor dado na árvore como se era uma árvore de busca binária desequilibrada, o que pode ocasionar o aumento da altura da sub-árvore onde o nó foi inserido, que por sua vez altera os fatores de balanceamento dos nós daquela sub-árvore.

Para manter a árvore balanceada a cada inserção devemos ter um algoritmo que efetue a inserção propriamente dita, ajustando os fatores de balanceamento para que eles se adaptem ao novo formato da árvore e verifique se houve quebra no equilíbrio da mesma. Se a árvore não estiver balanceada ou seja não estiver no intervalo [-1,+1], o algoritmo deve corrigir a sua estrutura através de movimentação dos nós, o que chamamos de rotação.

Caso haja quebra do balanceamento, podemos cair em 4 casos distintos, cujo tratamento para se re-estruturar a árvore é diferente:

Caso 01: Entrada [10, 30, 50]

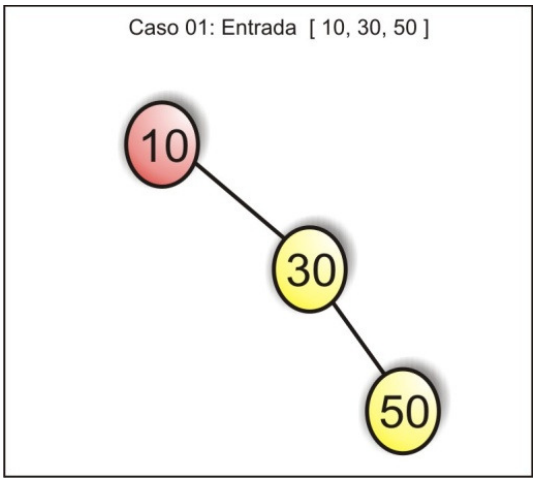


Figura 3.40: Inserção em uma árvore AVL 01.

Nesse caso, a ordem de entrada dos dados (crescente), faz com que a árvore resultante quebre o balanceamento, onde dada uma raiz *r*, a quebra do balanceamento tem como características:

- Todos os nós da sub-árvore de *r* em que ocorreu a quebra do balanceamento são maiores que *r*; e para uma raiz arbitrária *ra* dentro dessa sub-árvore, vale a relação recursiva desta mesma propriedade para *ra*;

- Em outras palavras: Dado uma raiz *r*, o filho de r sempre será maior que *r*;

- Logo, temos uma árvore inclinada para a direita.

Caso 02: Entrada [50, 30, 10]

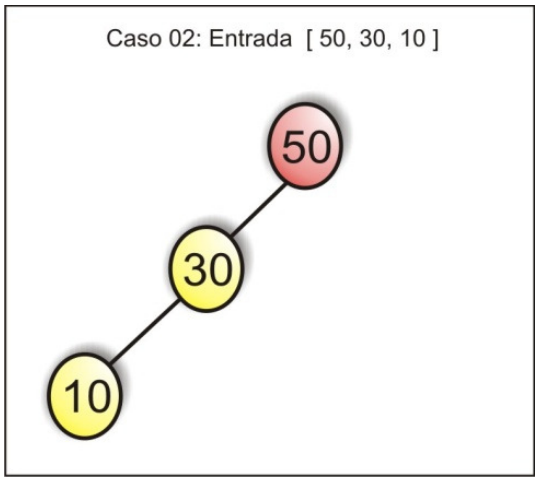


Figura 3.41: Inserção em uma árvore AVL 02.

Nesse caso, a ordem de entrada dos dados (decrescente), faz com que a árvore resultante quebre o balanceamento, onde dada uma raiz *r*, a quebra do balanceamento tem como características:

- Todos os nós da sub-árvore de *r* em que ocorreu a quebra do balanceamento são menores que *r*; e para uma raiz arbitrária *ra* dentro dessa sub-árvore, vale a relação recursiva desta mesma propriedade para *ra*;

- Em outras palavras: Dado uma raiz *r*, o filho de *r* sempre será menor que *r*;

- Logo, temos uma árvore inclinada para a esquerda.

Caso 03: Entrada [10, 50, 30]

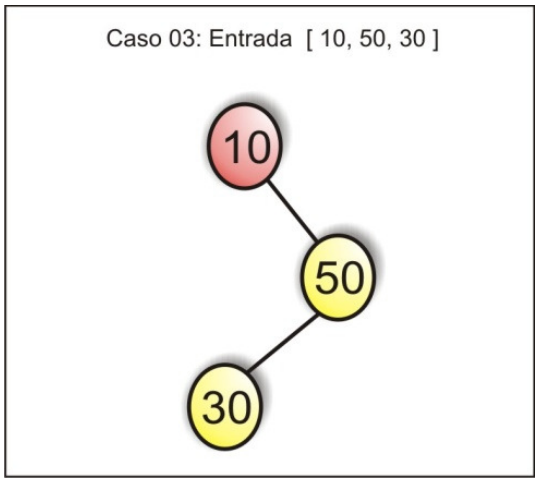


Figura 3.42 Inserção em uma árvore AVL 03.

Nesse caso, a ordem de entrada dos dados, faz com que a árvore resultante quebre o balanceamento, onde dada uma raiz *rp*, a quebra do balanceamento tem como características:

- Dada uma raiz *rp*, o filho de *rp* no qual a árvore quebrou o balanceamento é maior que *rp*; e considerando o filho de *rp* uma raiz *rs*, o filho de *rs* no qual a árvore quebrou o balanceamento é menor do que *rs*;

- A chave é inserida na sub-árvore direita da raiz(1), mas pesa a esquerda.

Caso 04: Entrada [50, 10, 30]

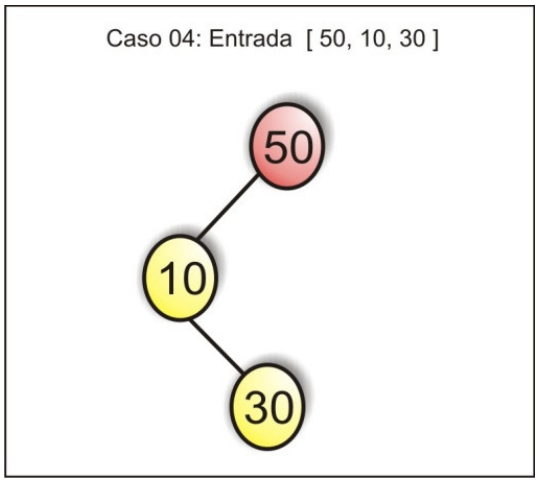


Figura 3.43: Inserção em uma árvore AVL 03.

Nesse caso, a ordem de entrada dos dados, faz com que a árvore resultante quebre o balanceamento, onde dada uma raiz *rp*, a quebra do balanceamento tem como características:

- Dada uma raiz *rp*, o filho de *rp* no qual a árvore quebrou o balanceamento é menor que *rp*; e considerando o filho de *rp* uma raiz *rs*, o filho de *rs* no qual a árvore quebrou o balanceamento é maior do que *rs*;

- A chave é inserida na sub-árvore esquerda da raiz(1), mas pesa a direita.

Embora existam quatro casos distintos, que exigem diferentes métodos para se balancear a árvore, o objetivo final da re-estruturação da árvore é sempre o mesmo, para tal utilizamos operações de rotações sobre os nós para resolver estes problemas.

Comumente quatro tipos de operações de rotação são realizadas nas árvores: rotação simples à esquerda, rotação simples à direita, rotação dupla à esquerda e rotação dupla à direita.

Caso 01: Rotação Simples à Esquerda

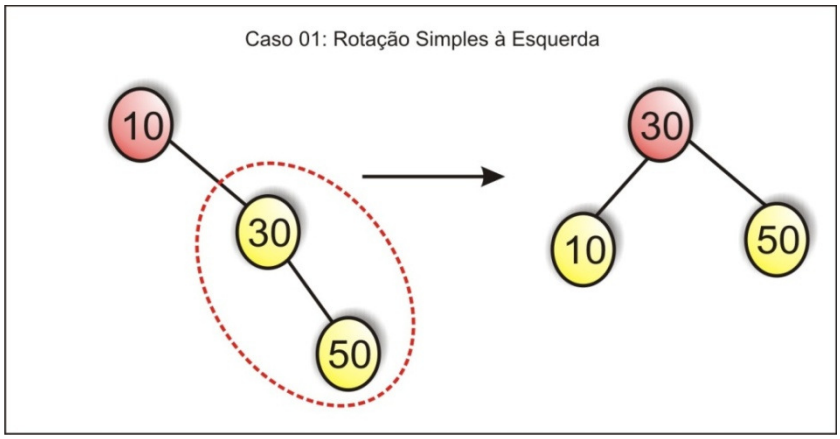


Figura 3.44: Rotação Simples à Esquerda.

- O nó intermediário(30) é maior que seu pai(10), porém, é menor que seu filho(50).

- Logo o nó intermediário(30) deve ser escolhido para ser a raiz da árvore resultante.

- O nó corrente(10) tem que cair para a esquerda, ou seja, ser rotacionado para esquerda.

Caso 02: Rotação Simples à Direita

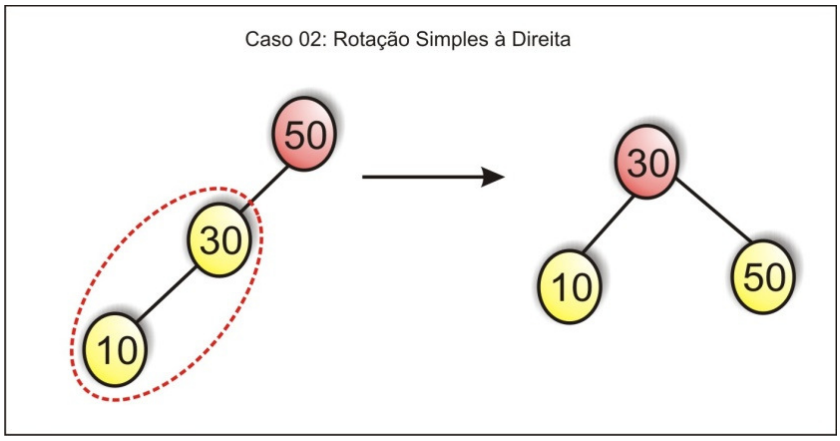


Figura 3.45: Rotação Simples à Direita.

- O nó intermediário(30) é menor que seu pai(50), porém, é maior que seu filho(10).

- Logo o nó intermediário(30) deve ser escolhido para ser a raiz da árvore resultante.

- O nó corrente(50) tem que cair para a direita, ou seja, ser rotacionado para direita.

Caso 03: Rotação Dupla à Esquerda

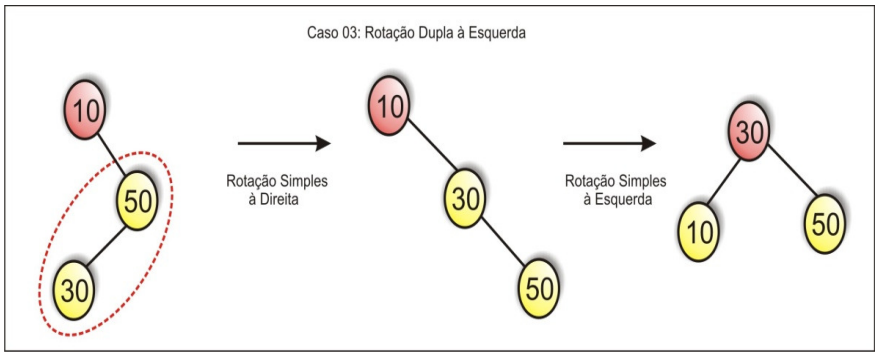


Figura 3.46: Rotação Dupla à Esquerda.

- O nó intermediário(50) é maior que seu pai(10), e é maior que seu filho(30).

- Logo o nó intermediário(50) deve ser trocado com o seu filho(30), por uma rotação à direita, caindo no 2º caso; Então aplica-se uma rotação à direita.

Caso 04: Rotação Dupla à Direita

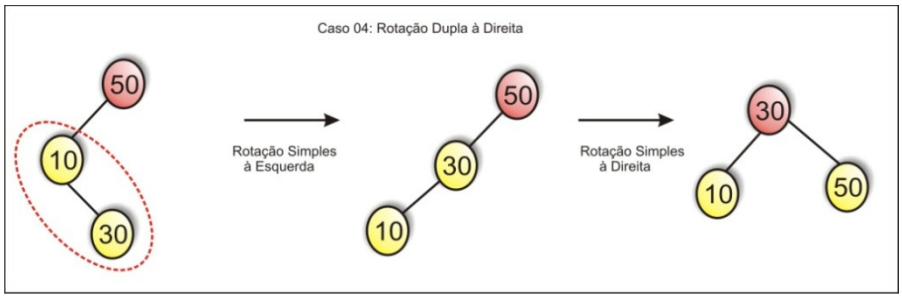


Figura 3.47: Rotação Dupla à Direita.

- O nó intermediário(10) é menor que seu pai(50), e é menor que seu filho(30).

- Logo o nó intermediário(10) deve ser trocado com o seu filho(30), por uma rotação à esquerda, caindo no 2º caso; Então aplica-se uma rotação à esquerda.

A figura 3.48 mostra a inserção de elementos em uma árvore AVL.

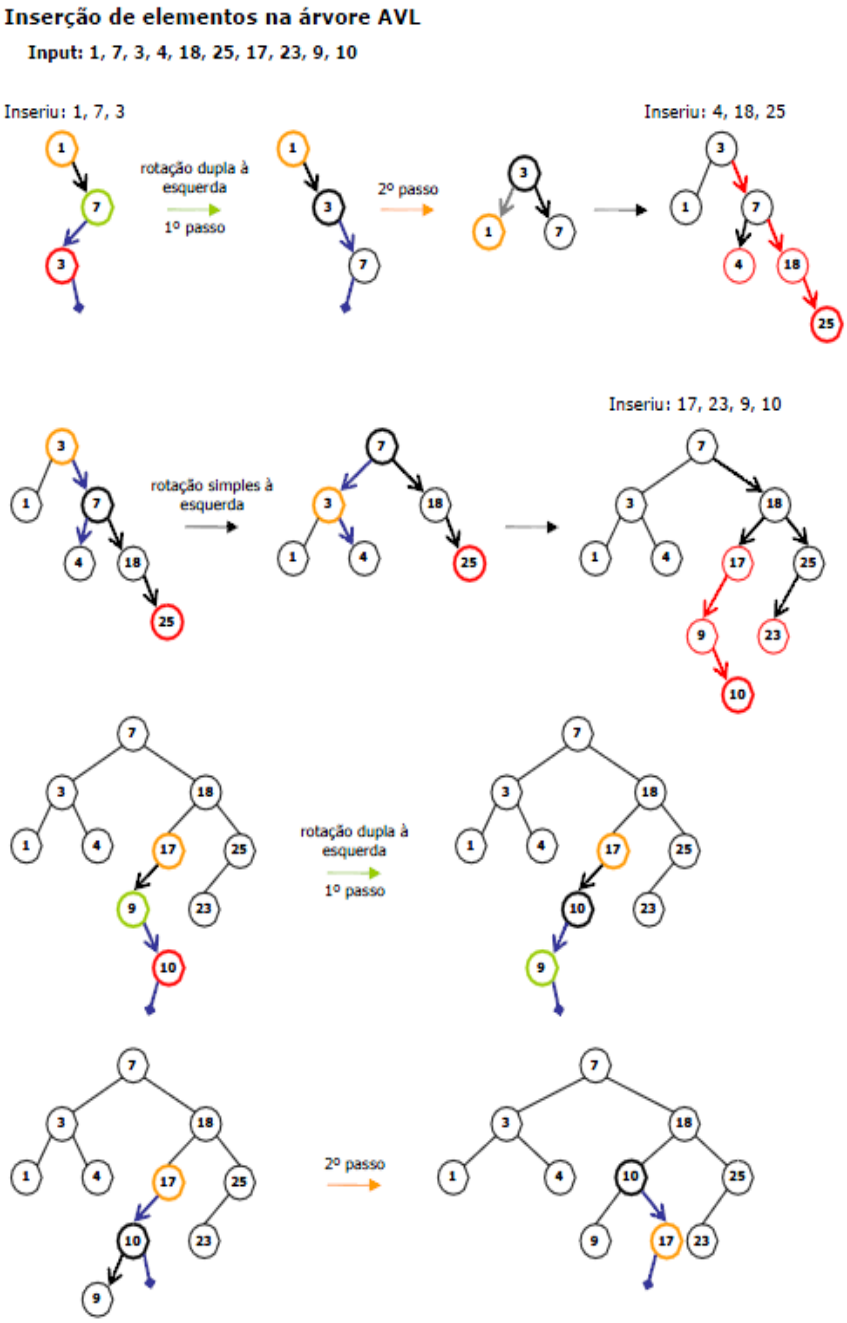


Figura 3.48: Inserção em uma árvore AVL.

A seguir é apresentado um pseudo-algoritmo para inserção de elementos em uma árvore AVL.

Insere AVL(Árvore T, chave)

/ Insere um nó na árvore procurando mantê-la balanceada */*

- Se vazia(T):

- Cria um novo nó e insere as informações do novo nó;

- Altura do novo nó é igual a 0;

- Retorna o endereço do nó criado;

- Compara a chave a ser inserida com a chave de r:

- Se igual, já está, retorna p;

- ***Se menor****:*

- Sub-árvore esquerda a r recebe o resultado de ***Insere(sub-árvore à esquerda a r, chave)****;*

- Calcula a diferença das alturas das sub-árvores esquerda e direita a r;

- ***Se*** *a diferença das alturas das sub-árvores for maior que 1,* ***houve quebra do balanceamento na árvore corrente****:*

- Se sub-árvore em que ocorreu a quebra do balanceamento for totalmente inclinada para a esquerda(ou seja, o nó foi inserido a esquerda):

- Retorna resultado de Rotacionar à direita;

- Se não, o nó foi inserido à direita:

- Rotaciona sub-árvore esquerda de r à esquerda;

- Retorna resultado de Rotaciona r à direita;

- Se não, não houve quebra do balanceamento:

- Calcula a altura de r baseado na altura de suas sub-árvores esquerda e direita;

- ***Se maior****:*

- Sub-árvore direita a r recebe o resultado de ***Insere(sub-árvore à direita a r, chave)****;*

- Calcula a diferença das alturas das sub-árvores esquerda e direita a r;

- ***Se*** *a diferença das alturas das sub-árvores for maior que 1,* ***houve quebra do balanceamento na árvore corrente****:*

- Se sub-árvore em que ocorreu a quebra do balanceamento for totalmente inclinada para a direita(ou seja, o nó foi inserido a direita):

- Retorna resultado de Rotaciona r à esquerda;

- Se não, o nó foi inserido à esquerda:

- Rotaciona sub-árvore direita de r à direita;

- Retorna resultado de Rotaciona r à esquerda;

- Se não, não houve quebra do balanceamento:

- Calcula a altura de r baseado na altura de suas sub-árvores esquerda e direita;

- Retorna r;

Praticar: Implemente um algoritmo para inclusão de dados em uma árvore AVL.

Remoção:

Quando retiramos um elemento da árvore binária, precisamos verificar se a remoção desse elemento quebrou a propriedade de balanceamento da árvore. Fazemos isto controlando as alturas das sub-árvores esquerda e direita da raiz da árvore. Se o módulo da diferença das alturas (altura da sub-árvore esquerda – altura da sub-árvore direita) for maior que 1, então houve a quebra do balanceamento, e há a necessidade de re-estruturar a árvore para que ela volte a ser balanceada.

No caso de quebra do balanceamento, podemos cair em 4 casos distintos, cujo tratamento para se re-estruturar a árvore é diferente:

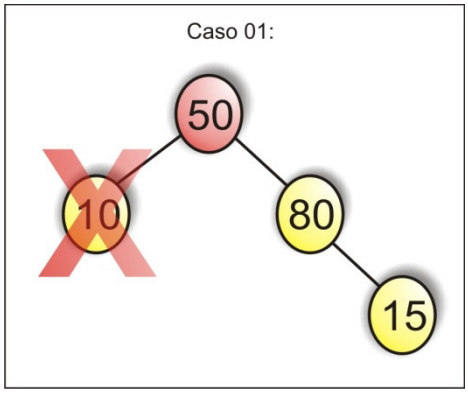


Figura 3.49: Remoção de elementos em uma árvore AVL 01.

Nesse caso, a remoção do nó 10 nos deixa uma árvore desbalanceada, do tipo 1, e então precisamos aplicar o algoritmo de rotação simples à esquerda sobre a raiz onde ocorreu a quebra do balanceamento.

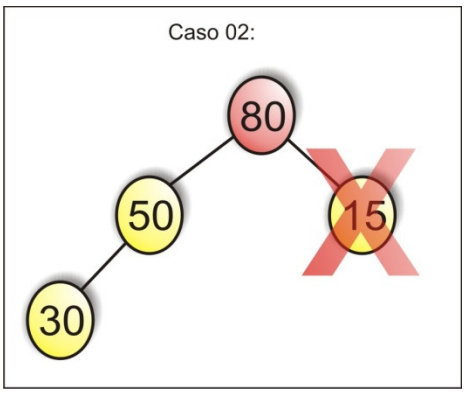


Figura 3.50: Remoção de elementos em uma árvore AVL 02.

Nesse caso, a remoção do nó 15 nos deixa uma árvore desbalanceada, do tipo 2, e então precisamos aplicar o algoritmo de rotação simples à direita sobre a raiz onde ocorreu a quebra do balanceamento.

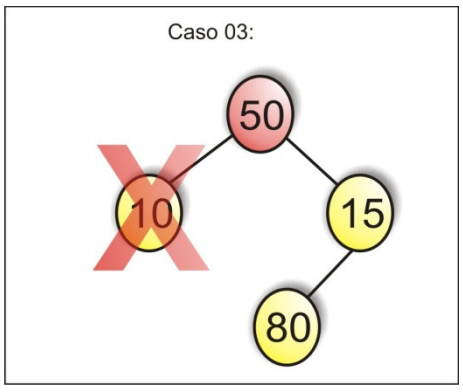


Figura 3.51: Remoção de elementos em uma árvore AVL 03.

Nesse caso, a remoção do nó 10 nos deixa uma árvore desbalanceada, do tipo 3, e então precisamos aplicar o algoritmo de

rotação dupla à esquerda sobre a raiz onde ocorreu a quebra do balanceamento.

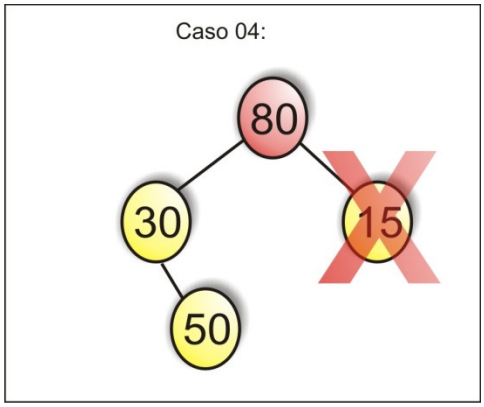


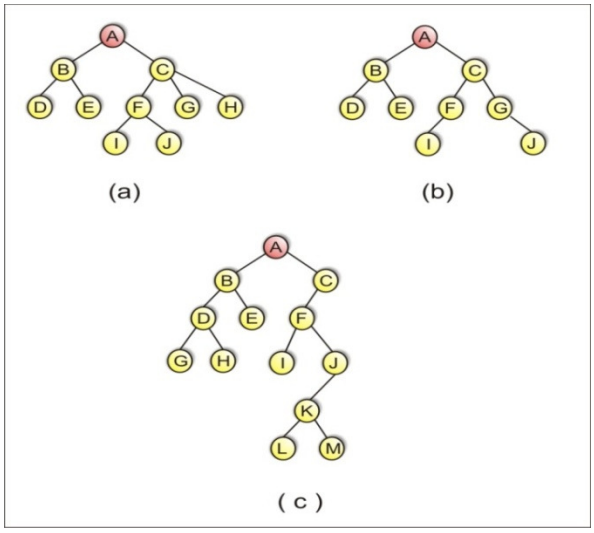
Figura 3.52: Remoção de elementos em uma árvore AVL 04.

Nesse caso, a remoção do nó 15 nos deixa uma árvore desbalanceada, do tipo 4, e então precisamos aplicar o algoritmo de rotação dupla à direita sobre a raiz onde ocorreu a quebra do balanceamento.

Praticar: Implemente um algoritmo, semelhante ao apresentado para inclusão, para remoção de dados em uma árvore AVL.

5.5 **Exercícios**

1. Verifique se as árvores abaixo estão ou não balanceadas. Em caso negativo (árvore não balanceada) balancei-as.



2. Monte uma árvore AVL utilizando o seguinte conjunto de dados [25, 58, 98, 41, 63, 17, 38, 26]. Realize as seguintes operações:

   a. Inclua em seqüência os elementos 12, 37 e 65;

   b. Remova os elementos 41, 17 e 26;

3. Implemente, em Java, um programa que realize as operações de inserção e remoção em árvores AVL.

UNIDADE II
ÁRVORES

6. OUTROS TIPOS DE ÁRVORES

6.1 ÁRVORES B

As árvores B são árvores de busca balanceadas projetadas para funcionar bem em discos magnéticos ou outros dispositivos de armazenamento secundário de acesso direto. As árvores B são semelhantes às árvores AVL, mas são melhores para minimizar operações de E/S de disco pois o disco é prejudicado principalmente pelo tempo de posicionamento do braço de leitura. Uma vez que o braço esteja posicionado no local correto, a leitura pode ser feita de forma bastante rápida. Desta forma, devemos minimizar o número de acessos ao disco.

As árvores B diferem significativamente das árvores AVL pelo fato de que os nós de árvores B podem ter muitos filhos(semelhante a árvores n-árias), isto é, o fator de ramificação de uma árvore B pode ser bastante grande, embora seja limitado por características do sistema operacional e o dispositivo secundário de armazenamento utilizado. As árvores B são semelhantes às outras árvores balanceadas no fato de que toda árvore B de n nós tem altura $O(\log n)$, embora essa altura possa ser consideravelmente menor que a altura de uma árvore AVL devido ao seu fator de ramificação, que é muito maior. Portanto, as árvores B podem ser utilizadas para implementar muitas operações no tempo $O(\log n)$.

Árvores B generalizam árvores de busca binária de maneira natural. Se um nó interno x de uma árvore B contém n chaves, então

x tem *n+1* filhos. Cada chave em *x* é utilizada como ponto de divisão que separam o nó *x* em *n+1* intervalos para a tomada de decisão, ou seja, em um nó *x* com n chaves, temos *n+1* possibilidades de decisão para qual ramo devemos seguir em frente, por exemplo, em uma busca de uma determinada chave. Quando procuramos por uma chave em uma árvore B, tomamos uma decisão de *n+1* modos, com base em comparações com as *n* chaves armazenadas no nó *x* e a chave desejada.

Suponhamos que quaisquer "informações extras" (informações diferente do atributo chave) associadas a uma chave estão armazenadas no mesmo nó em que está a chave. Na prática, seria possível armazenar com cada chave apenas uma referência para outra página de disco contendo as informações extras correspondentes a cada chave. A figura 3.53 ilustra um exemplo de uma árvore B.

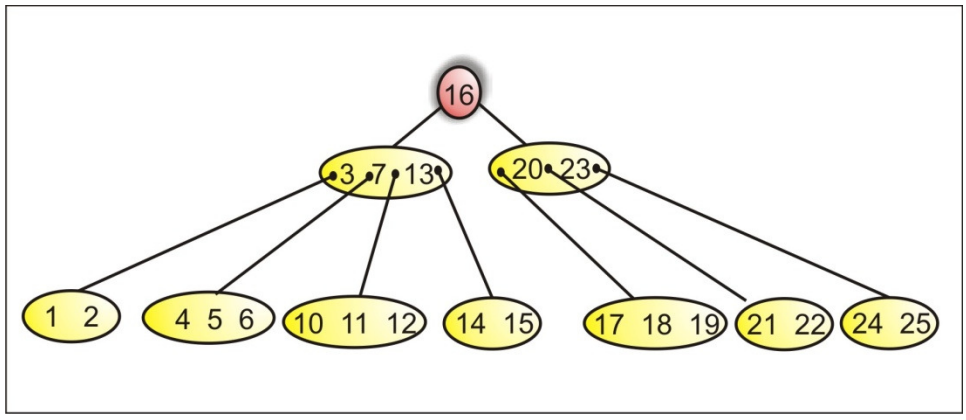


Figura 3.53: Árvore B.

Altura de uma árvore B

O número de acessos ao disco exigidos para a maioria das operações em árvore B é proporcional à altura da árvore B. Agora, vamos analisar a altura de uma árvore B no pior caso:

Se n≥1, então, para qualquer arvore B T de n nós de altura h e ordem d≥2: h≤log[(n+1)/2] na base d.

Vemos então a capacidade de armazenamento das árvores B, quando comparada a outras árvores balanceadas. Embora a altura da árvore cresça na proporção de *O(ln n)* em ambos os casos, para as árvores B, a base do logaritmo pode ser muitas vezes maior, diminuindo assim a quantidade de acessos ao disco para ser realizar uma operação de busca, por exemplo.

Operações: Busca

Busca em uma Árvore-B é um procedimento parecido com o de busca em árvore de busca binária, exceto que devemos decidir entre vários caminhos. Como as chaves estão ordenadas, basta realizarmos uma busca binária nos elementos de cada nó que leva tempo *O(lg(t))*, se o elemento não for encontrado naquele nó vamos para o filho apropriado e realizamos a busca binária. Desta forma a busca completa pode ser realizada em tempo *O(lg(t)logt(n))*.

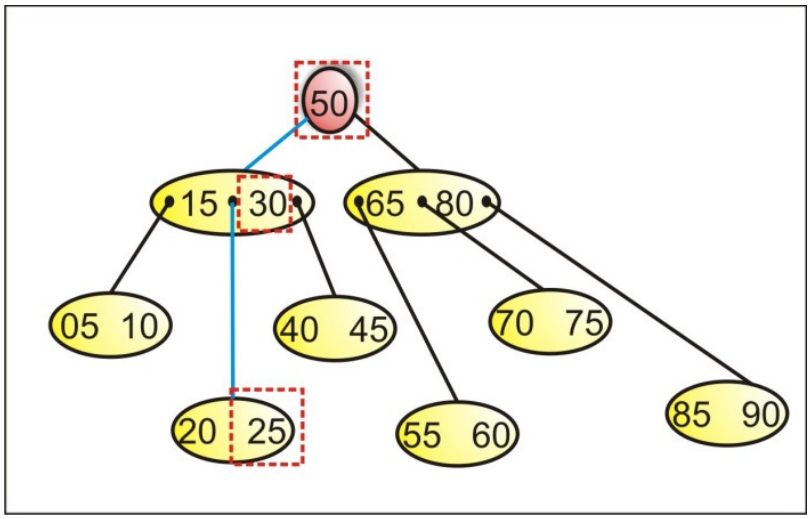


Figura 3.54: Operação de busca em Árvore B.

Operações: Inclusão

Para inserirmos um novo elemento em uma Árvore-B, basta localizar o nó folha *X* onde o novo elemento deva ser inserido. Se o nó *X* estiver cheio, precisamos realizar uma subdivisão de nós, que consiste em passar o elemento mediano de *X* para seu pai e subdividir *X* em dois novos nós com *t-1* elementos e depois inserir a nova chave.

Se o pai de *X* também estiver cheio, repetimos recursivamente a sub-divisão acima para o pai de *X*. No pior caso, teremos que aumentar a altura da Árvore-B para podermos inserir o novo elemento. Note que diferente das árvores binárias, as Árvores-B crescem para cima. A figura 3.55 ilustra a inclusão de novos elementos em uma Árvore-B.

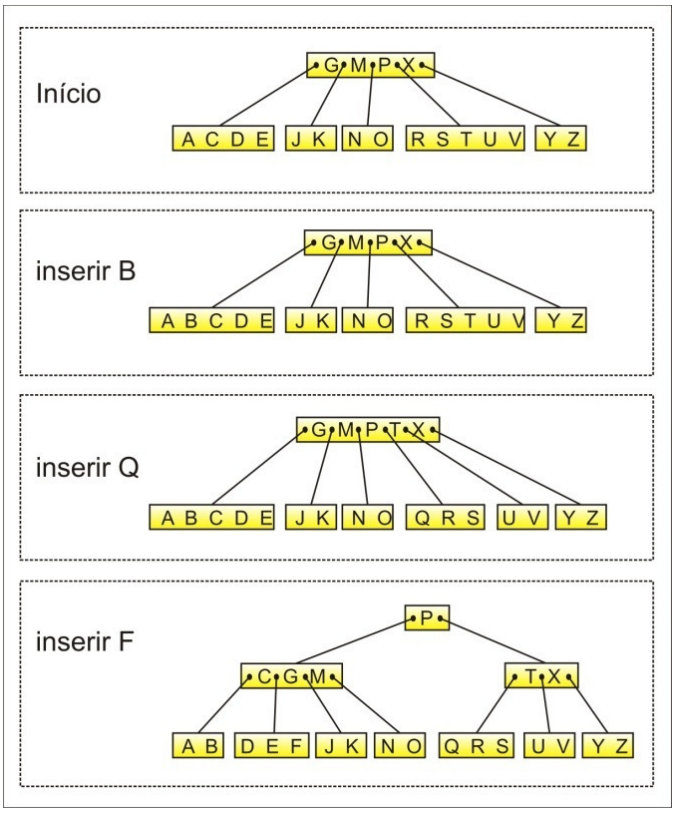


Figura 3.55: Operação de inclusão em Árvore B.

Operações: Remoção

A remoção de um elemento de uma Árvore-B pode ser dividida em dois caso:

1. O elemento que será removido está em uma folha
2. O elemento que será removido está em um nó interno.

Se o elemento estiver sendo removido de um nó não folha, seu sucessor, que deve estar em uma folha, será movido para a posição eliminada e o processo de eliminação procede como se o elemento sucessor fosse removido do nó folha.

Quando um elemento for removido de uma folha *X* e o número de elementos no nó folha cai para menos que *t-1*, devemos reorganizar a Árvore-B. A solução mais simples é analisarmos os irmão da direita ou esquerda de *X*. Se um dos irmãos (da direita ou esquerda) de *X* possui mais de *t-1* elementos, a chave *k* do pai que separa os irmãos pode ser incluída no nó *X* e a última ou primeira chave do irmão (última se o irmão for da esquerda, primeira se o irmão é da direita) pode ser inserida no pai no lugar de *k*.

Se os dois irmãos de *X* contiverem exatamente *t-1* elementos, nenhum elemento poderá ser deslocado. Neste caso, o nó *X* e um de seus irmãos são concatenados em um único nó que contém também a chave separadora do pai.

Se o pai também contiver apenas *t-1* elementos, devemos considerar os irmãos do Pai como no caso anterior e procedermos recursivamente. No pior caso, quando todos os ancestrais de um nó e seus irmãos contiverem exatamente *t-1* elementos, uma chave será tomada da raiz e no caso da raiz possuir apenas um elemento a Árvore-B sofrerá uma redução de altura.

6.2 ÁRVORES B+

Árvores B+ são como uma extensão do conceito de árvores B, embora conceitualmente sejam muito parecidas. A árvore B+ é uma estrutura um pouco diferente da árvore B, que além de proporcionar acesso indexado aos dados, também possibilita acesso seqüencial. Outra característica que difere esta variação de árvores B é a maximização da quantidade de registros que cada nó pode suportar, pois mantêm apenas o atributo determinante (“chave”) de cada registro no nó, guardando consigo apenas um ponteiro para as outras informações do registro. Ou seja, um nó de uma árvore B+ não armazena os dados de cada registro dentro de seus nós, sendo estes armazenados fora da estrutura da árvore ou apenas nas folhas da árvore (embora na prática, a primeira solução seja a mais utilizada por SGBDS – Sistemas Gerenciais de Bancos de Dados).

Esta forma diferente de se armazenar os registros, não maximiza apenas a quantidade de registros que um nó pode conter, como também traz uma série de outras vantagens. Podemos agora, com a árvore B+, criar vários índices para um mesmo grupo de registros, já que as informações não estão mais inerentes à estrutura do nó da árvore, e estão disponíveis para serem manipulados livremente. Isto possibilita que as árvores B funcionem como índices baseados em atributos distintos do registro para um mesmo grupo de registros. Por exemplo, em um sistema de uma universidade, poderíamos criar índices para fazer consultas às informações dos alunos baseados em matricula, e outro em nome do aluno, entretanto ambos os índices se referem ao mesmo conjunto de dados.

Concluindo, as árvores B+ são ótimas para se implementar índices para conjuntos de registros. Normalmente árvores que funcionam como índices, os nós só possuem, além da chave, um ponteiro para o registro de dados em outro arquivo. Isto pode ser

levado ao extremo, se nós concentramos os ponteiros para o arquivo de registros nas folhas, e é isto que a implementação de árvore B+ vindoura faz.

Característica

- Acesso indexado: Os nós internos servem só como referência para o percurso, criando um mecanismo para recuperar um registro sem que seja necessário percorrer todos os registros do arquivo de registros de dados. - Chaves de nós internos são repetidas nas folhas.

- O ponteiro para o registro de dados de uma chave somente se encontra no nó folha em que esta chave se encontra.

- A árvore é dividida em dois tipos de nós: nós internos (*Index Set)* e folhas (*Sequence Set).*

- Acesso Seqüencial: Os nós folhas (*Sequence Set*) são encadeados entre si, formando uma espécie de lista no nível das folhas. Isto nos dá um mecanismo para percorrer seqüencialmente o arquivo de registros sem que seja necessário ordenar os registros de dados. Isto é muito desejável na maioria das aplicações.

Estrutura

Como mencionado anteriormente, os nós internos de uma árvore B+ são diferentes de seus nós folhas: Os nós folhas das árvores B+ contêm além da chave, um ponteiro para o registro de dados respectivo a chave dada. Portanto ao manipularmos os nós folhas, temos de ter um tratamento diferenciado ao que demos às árvores B; enquanto que em relação aos nós internos, não há

diferença de manipulação. Os nós folhas se diferem basicamente por não terem *2d+1* (desconsiderando o nó extra que facilita a implementação - veja árvores B) ponteiros para outros nós, mas, no lugar destes, têm *2d* ponteiros para *2d* registros que estão armazenados à parte da estrutura da árvore.

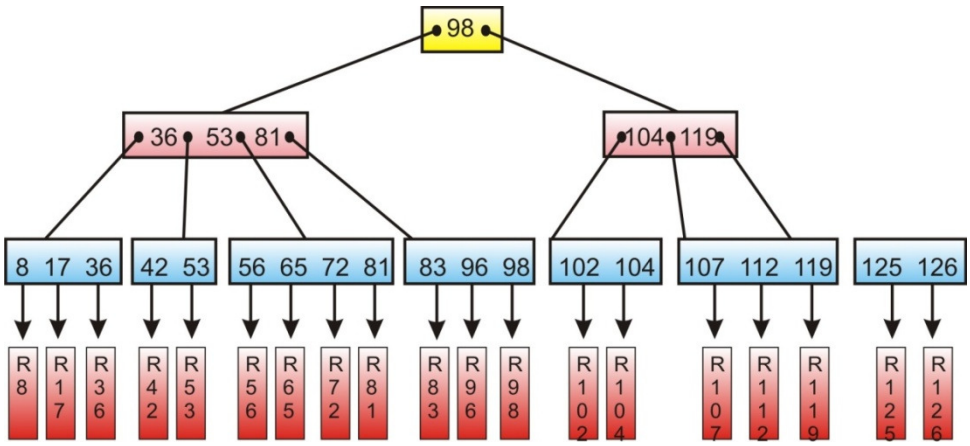


Figura 3.56: Operação de inclusão em Árvore B.

É importante esta separação lógica da árvore-B+ em Conjunto de Índices (*Index Set*) e Conjunto de Seqüência,
pois podemos fazer a maioria das inclusões e exclusões no conjunto de seqüência sem alterar o índice (árvore-B).
Quando houver necessidade de inclusão ou exclusão do referido índice, usamos os algoritmos já conhecidos da árvore-B.

A inserção numa árvore-B+ ocorre praticamente da mesma forma que numa árvore-B, exceto pelo fato de que, quando um nó é dividido, a chave do meio é retida no meio nó esquerdo, além de ser promovida a pai. Quando uma chave é eliminada de uma folha, ela pode ser retida nas não-folhas porque ela ainda é um separador válido entre as chaves nos nós abaixo dela.

A árvore-B+ mantém o baixo custo das operações de pesquisa, inclusão e exclusão e adquire a vantagem de requerer no máximo um acesso para satisfazer a operação da próxima chave.

Além disso, para um processamento seqüencial nenhum nodo precisará ser acessado mais de uma vez como acontece no caminhamento *in-order* que precisamos fazer numa árvore-B. Portanto, enquanto a eficiência de localização do registro seguinte em um árvore-B é *O(log n),* numa árvore-B+ é aumentada para *O(1).*

A árvore-B+ é ideal para aplicações que requerem tanto acesso seqüencial quanto aleatório. Por isso e pelas vantagens citadas anteriormente, a árvore B+ tornou se bastante popular nos SGBDs comerciais.

6.2 ÁRVORES B* (B Star)

Existem várias formas de aprimorar a utilização do armazenamento de uma árvore-B. Um método é retardar a divisão de um nó no caso de encher. Em substituição, as chaves no nó e um de seus irmãos adjacentes, bem como a chave no pai que separa entre os dois nós, são redistribuídas uniformemente. Esse processo é ilustrado na figura 3.57, numa árvore-B de ordem 7.

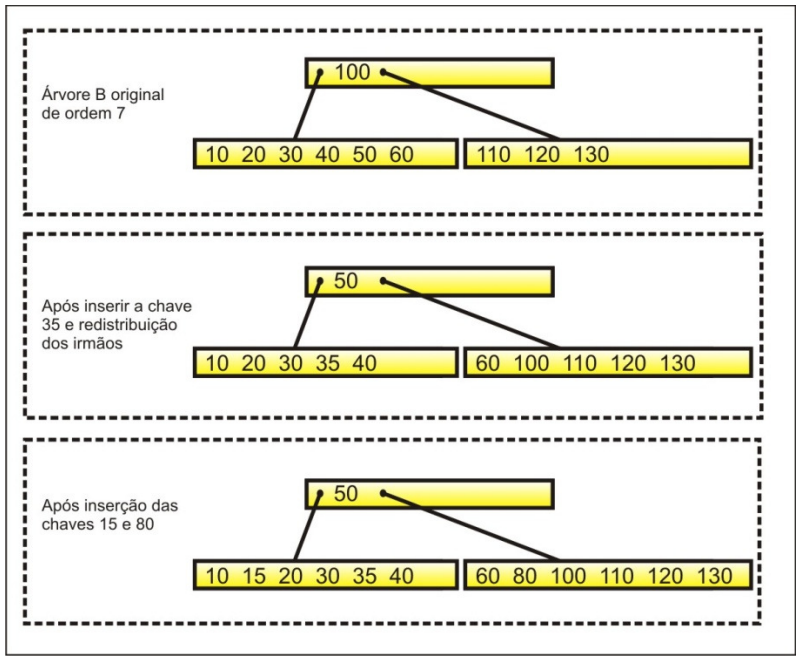


Figura 3.57: Árvore B*.

Quando um nó e seu irmão estiverem completos, os dois nós serão divididos em três (*two-to-three splitting*). Isso assegurará uma utilização mínima do armazenamento de quase 67%, contra 50% de uma árvore-B tradicional. Na prática, a utilização do armazenamento será ainda mais alta. Esse tipo de árvore é chamado de árvore-B*.

Essa técnica pode ser ainda mais ampliada redistribuindo as chaves entre todos os irmãos e o pai de um nó completo. Infelizmente, este método impõe um preço porque exige espaços adicionais dispendiosos durante as inserções, enquanto a utilização do espaço adicional alcançado pela consideração de cada irmão extra torna-se cada vez menor.

6.3 Exercícios

Aprofunde seus estudos em árvores B, B+ e B*. Verifique as principais operações relacionadas a elas e elabore um relatório sobre tal. O relatório deve conter no máximo 20 páginas e deve ser escrito no padrão de artigo científico da SBC - Sociedade Brasileira de Computação. O modelo está disponível em www.sbc.org.br

7. WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI
www.ufpi.br/uapi

Universidade Aberta do Brasil – UAB
www.uab.gov.br

Secretaria de Educação à Distância do MEC - SEED
www.seed.mec.gov.br

Associação Brasileira de Educação à Distância – ABED
www.abed.org.br

Curso de Estruturas de Dados: USP/São Carlos.
Prof. Graça Pimentel, Maria Cristina e Rosane
http://www.icmc.usp.br/~sce182/index.html

Curso de Estruturas de Dados: UNISINOS.
Prof. Leandro Tonietto
http://inf.unisinos.br/~ltonietto/jed/aed/aed2007_01.html

Curso de Estruturas de Dados: IME/USP.
Prof. Paulo Feofiloff
http://www.ime.usp.br/~pf/algoritmos/

Curso de Estruturas de Dados: UFPB.
Prof. Liliane dos Santos Machado
http://www.di.ufpb.br/liliane/aulas/index_ED.html

Algoritmos Animados em Java
http://gdias.artinova.pt/projecto/pt/menu_applets.php

Applet Árvore Genérica
http://www.engin.umd.umich.edu/CIS/course.des/cis350/treetool/index.html

Applets Árvore de Pesquisa
http://mainline.brynmawr.edu/Courses/cs206/spring2004/WorkshopApplets/Chap08/Tree/Tree.html

Applets Árvore de Pesquisa
http://www.cs.jhu.edu/%7Egoodrich/dsa/trees/btree.html

Applets Árvore de Pesquisa
http://www.ibr.cs.tubs.de/courses/ss98/audii/applets/BST/BST-Example.html

Applets Árvore AVL
http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm

Applet Árvore B
http://slady.net/java/bt/view.php?w=600&h=450

8. REFERÊNCIAS BIBLIOGRÁFICAS

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C. , ***Algoritmos***, Ed. Campus, 2002.

DROZDEK, A. , ***Estruturas de Dados e Algoritmos em C++***, Ed. Thomson, 2002.

GOODRICH, M. T. & TAMASSIA, R. , ***Estruturas de Dados e Algoritmos em Java***, Ed. Bookman, 2007.

LAFORE, R. , ***Estruturas de Dados e Algoritmos em Java***, Ed. Ciência Moderna, 2004.

LAUREANO, M. ***, Estrutura de Dados com Algoritmos e C***, Ed. Brasport, 2008.

LORENZI , F. & MATTOS , P. N. & CARVALHO, T. , ***Estruturas de Dados***, Ed. Cengage Learning, 2006.

PEREIRA***, Estruturas de Dados Fundamentais: Conceitos e Aplicações***, Ed. Érica, 2006.

PREISS, B. R. , ***Estruturas de Dados e Algoritmos***, Ed. Campus, 2000.

PUGA, S. & RISSETI, G. , ***Lógica de Programação e Estruturas de Dados***, Ed. Prentice-Hall, 2004.

RANGEL, J. L. & CERQUEIRA, R. & CELES, W. , ***Introdução a Estrutura de Dados***, Ed. Campus, 2004.

SCHILDT, H. , ***C Completo e Total***, Ed. Makron Books, 1996.

SILVA, O. Q. , ***Estruturas de Dados e Algoritmos Usando C: Fundamentos e Aplicações***, Ed. Ciência Moderna, 2007.

SZWARCFITER, J. L. & MARKENZON, L. , ***Estruturas de Dados e Seus Algoritmos***, Ed. LTC, 1994.

TENEMBAUM, A. M. ***, Estruturas de Dados Usando C***, Ed. Makron Books, 1995.

VELOSO, P. A. S. , ***Estruturas de Dados***. Ed Campus, 1983.

WIRTH, N. , ***Algoritmos e Estruturas de Dados***, Ed. LTC, 1989.

ZIVIANI, N. . ***Projeto de Algoritmos com implementação em PASCAL e C***, Ed. Thomson, 2005.

Esta unidade apresenta uma breve introdução sobre a Teoria dos Grafos. A teoria dos grafos estuda objetos combinatórios, os grafos, que são bons modelos para muitos problemas em vários ramos da matemática, da informática, da engenharia e da indústria.

Abordaremos mais fortemente os conceitos teóricos relacionados aos grafos com o intuito de proporcionar ao leitor um forte embasamento e encorajá-lo a mergulhar nesta área. Será apresentado também os problemas clássicos relacionados aos grafos, pois muitos destes problemas tornaram-se célebres porque são um interessante desafio intelectual e porque têm importantes aplicações práticas.

Esperamos que ao final o leitor tenha se convencido da utilidade dos conceitos e processos apresentados, mas guardamos o secreto desejo de que os aspectos lúdicos dos grafos o contaminem com o que costumamos chamar de ”*graphical desease*”, ou melhor, traduzindo, a febre dos grafos.

Cada capítulo é acompanhado de exercícios, sem a solução, preferimos deixar o prazer desta tarefa ao leitor. A bibliografia e a webliografia ao fim das notas são mais do que suficiente para adquirir um conhecimento razoável de teoria dos grafos.

SUMÁRIO

UNIDADE III – INTRODUÇÃO AOS GRAFOS

UNIDADE III
INTRODUÇÃO AOS GRAFOS

4. INTRODUÇÃO

1.1. Breve Histórico

Os primeiros trabalhos em teoria dos grafos surgiram no século XVIII. Vários autores publicaram artigos neste período, com destaque para o problema descrito por Euler, conhecido como “As Pontes de Königsberg”.

• As Pontes de Königsberg – Este problema foi proposto em um artigo publicado em 1736. Numa cidade chamada Königsberg, sete pontes estabelecem ligações entre as margens do Rio Pregel e duas de suas ilhas. O problema consiste em, a partir de um determinado ponto, passar por todas as pontes somente uma vez e retornar ao ponto inicial. Esse problema ficou conhecido como “Problema do caminho Euleriano” e será estudado mais adiante. A figura 4.1 ilustra o problemas das Pontes de Königsberg.

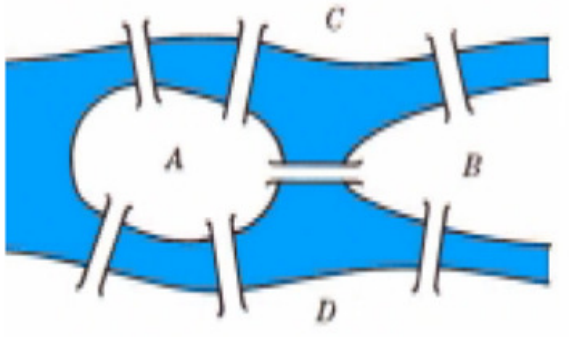


Figura 4.1: Pontes de Königsberg.

Euler resolveu este problema criando um grafo em que terra firme é vértice e ponte é aresta. Ele procurava uma seqüência da forma $R_0, P_1, R_1, P_2, R_2, P_3, R_3, P_4, R_4, P_5, R_5, P_6, R_6, P_7, R_7$ onde:

(a) $R_0, R_1, ..., R_7$ são regiões;
(b) P_i é uma ponte ligando R_{i+1} a R_i.
(c) $P_0, P_1, ..., P_7$ são diferentes.

A representação por ele utilizada pode ser vista na figura 4.2.

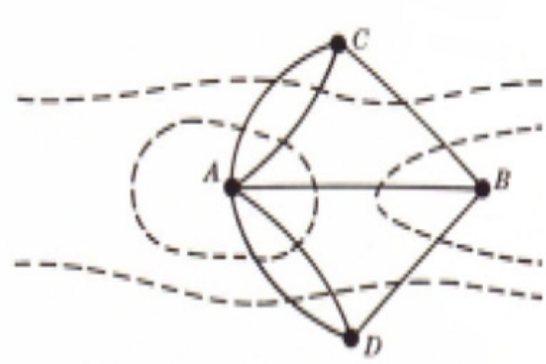


Figura 1.2: Representação em um grafo do problema da Pontes de Königsberg.

Euler chegou à conclusão que era impossível encontrar essa seqüência. Pois, quando caminhamos por um vértice, nós temos que entrar e sair dele (ou vice-versa, no caso do ponto inicial), o que significa que usamos um número par de arestas cada vez que passamos por um vértice. Como o grafo acima possui vértices com número ímpar de arestas, a resposta para o problema é NÃO.

Outro grande pesquisador foi o alemão Kirchhoff que em 1847 utilizou grafos para modelagem de circuitos elétricos.

Ford e Fulkerson (1962) desenvolveram a teoria dos fluxos em redes, um dos mais importantes resultados da teoria dos grafos, e muitas outras aplicações da teoria dos grafos então sendo desenvolvidas na área de Pesquisa Operacional.

1.2. Conceitos iniciais

Considere que em uma escola algumas turmas resolveram realizar um torneio de futebol. Participam do torneio as turmas *6A, 6B, 7A, 7B, 8A e 8B*. Alguns jogos foram realizados até agora:

6A jogou com 7A, 7B, 8B
6B jogou com 7A, 8A, 8B
7A jogou com 6A, 6B
7B jogou com 6A, 8A, 8B
8A jogou com 6B, 7B, 8B
8B jogou com 6A, 6B, 7B, 8A

Mas será que isto está correto? Pode ter havido um erro na listagem. Uma maneira de representar a situação através de uma figura é a seguinte (figura 4.3):

- as turmas serão representadas por pontos;
- e os jogos serão representados por linhas;

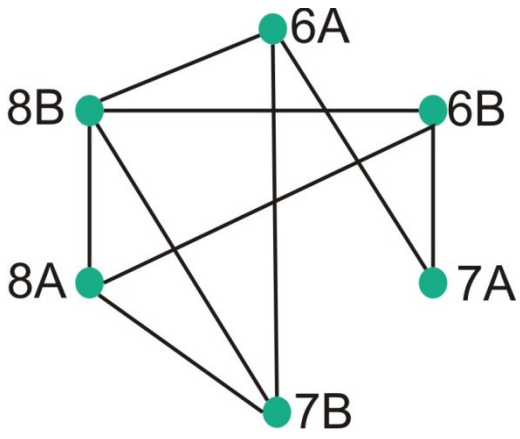


Figura 4.3: Representação de um campeonato de futebol na forma de um grafo.

Não é difícil agora constatar a consistência das informações. A estrutura que acabamos de conhecer é um grafo. Em outras palavras, um grafo é um modelo matemático que representa relações entre objetos.

Para que um grafo fique bem definido temos que ter dois conjuntos:

- O conjunto *V*, dos vértices (no nosso exemplo, o conjunto das turmas).

- O conjunto *A*, das arestas **(**no nosso exemplo, os jogos realizados).

Quando existe uma aresta ligando dois vértices dizemos que os vértices são adjacentes e que a aresta é incidente aos vértices. No nosso exemplo podemos representar o grafo de forma sucinta como:

V = {6A; 6B; 7A; 7B; 8A; 8B} e
A = {(6A; 7A); (6A; 7B); (6A; 8B); (6B; 7A); (6B; 8A);
(6B; 8B); (7B; 8A); (7B; 8B); (8A; 8B)}

Observe que não precisamos colocar (8*A*; 7*B*) no conjunto de arestas, pois já tínhamos colocado (7*B*; 8*A*).

O número de vértices será simbolizado por |*V*| ou pela letra *n*. O número de arestas será simbolizado por |*A*| ou pela letra *m*. No nosso exemplo *n* = 6 e *m* = 9.

Ainda no nosso exemplo vimos que cada turma jogou um número diferente de jogos:

6A jogou 3 jogos
6B jogou 3 jogos
7A jogou 2 jogos
7B jogou 3 jogos
8A jogou 3 jogos
8B jogou 4 jogos

Por isso, no nosso desenho, o vértice 6*A* tem 3 arestas ligadas a ele, o vértice *A*7 tem 2 arestas ligadas a ele e assim por diante. Dizemos que estas arestas são incidentes ao vértice. O número de vezes que as arestas incidem sobre o vértice *v* é chamado grau do vértice *v*, simbolizado por *d*(*v*). No nosso exemplo, *d*(6*A*) = 3; *d*(7*A*) = 2.

Praticar: Usando o grafo do campeonato dê o grau de cada um dos vértices.

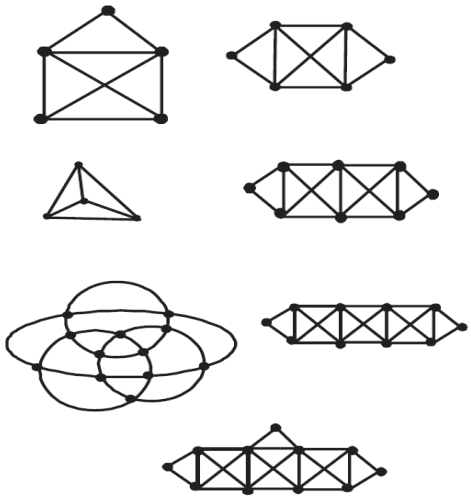
1.3. Exercícios

1 - Considere um campeonato de futebol formado por:

Equipe 01: Matemática;
Equipe 02: Física;
Equipe 03: Química;
Equipe 04: Biologia;
Equipe 05: Computação1;

Esboce um grafo que represente todas as combinações de jogos possíveis entre essas equipes. Utilizando este mesmo grafo dê o grau de cada vértice e a quantidade de arestas nele existente.

2 – Dê o grau de cada vértice e o número de arestas para cada um dos grafos abaixo.



UNIDADE III
INTRODUÇÃO AOS GRAFOS

5. DEFINIÇÕES

São muitos os conceitos relacionais à teoria dos grafos e aqui apresentaremos as que consideramos básicas para um estudo inicial sobre o tema.

2.1. Definições básicas

Grafo: Um grafo *G(V,E)* é uma estrutura constituída por um conjunto finito não vazio de vértices ou pontos v_1, v_2,...,v_n (denotado conjunto *V*) e um conjunto de linhas ,ramos ou arestas e_1, e_2,...,e_n (denotado conjunto *E*) unindo todos ou alguns desses pontos.

Representação Geométrica: Normalmente um grafo pode ser visualizado através de uma representação geométrica, na qual seus vértices correspondem a pontos distintos do plano em posições arbitrárias, enquanto que a cada aresta *(v, w)* é associada uma linha arbitrária unindo os pontos correspondentes a *v* e *w.* A figura 4.4 apresenta um grafo *G(V,E)* onde o conjunto de vértices V é {1,2,3,4,5,6} ou V = {1,2,3,4,5,6} e o conjunto de arestas é E = { *(1,2),(1,3),(2,4),(3,4),(4,5),(5,6)*}.

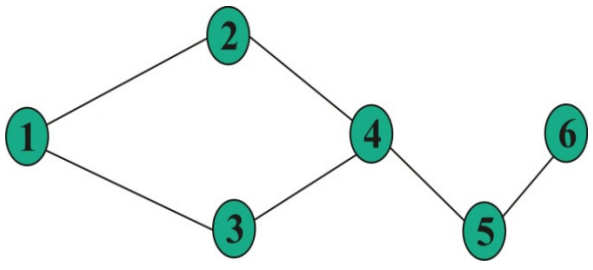


Figura 4.4: Grafo com seis vértice e seis arestas.

Arestas incidentes e adjacentes: Cada aresta de um grafo é denotada por *e = (v, w)* e é composta pelo par de vértices que a forma. Neste caso, os vértices *v* e *w* são os extremos da aresta *e*. Uma aresta se diz incidente em um vértice *v* se *v* é um de seus extremos e os vértices *v* e *w* se dizem adjacentes se existe uma aresta que incide em ambos. A figura 4.5 ilustra os conceitos apresentados.

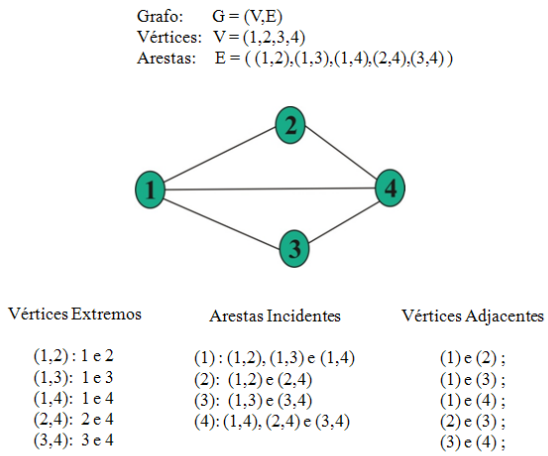


Figura 4.5: Vértices externos, arestas incidentes e vértices adjacentes.

Interpretando a figura tem-se que em relação à aresta (1,2) os vértices (1) e (2) são externos a ela, que as arestas incidentes no vértice (3) são (1,2) e (2,4) e que os vértices (2) e (3) são adjacentes, por exemplo.

Grafo direcional: Um grafo é dito direcional, ou dígrafo, quando é necessário ser estabelecido um sentido (orientação) para as arestas. O sentido da aresta é indicado através de uma seta, como ilustrado na figura 4.6. Nesta situação, a aresta passa a ser chamada de arco.

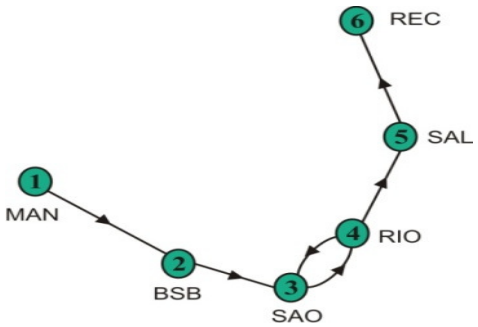


Figura 4.6: Grafo direcional ou dígrafo.

Algumas vezes é útil relaxar a definição de grafos, de modo a permitir uma aresta do tipo *e = (v,v).* Uma aresta desta natureza é chamada de laço. Outra extensão possível consiste em substituir, na definição do grafo, o conjunto de arestas *E* por um multiconjunto de forma a permitir a existência de mais de uma aresta entre os mesmos vértices. Estas arestas são chamadas paralelas. Um exemplo de grafo que contém arestas de laço é apresentado na figura 4.7(a) e um grafo que contém arestas paralelas em 4.7(b).

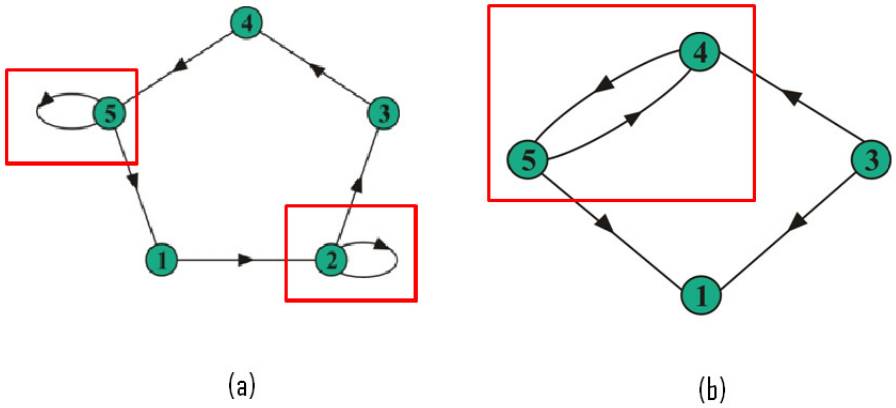


Figura 4.7: (a) Grafo com arestas de laço (b) grafo com arestas paralelas.

Grau, Sucessores e Antecessores: Em um grafo *G*, define-se grau de um vértice $v_i \in V$, denotado por $grau(v_i)$ como sendo o número de vértices adjacentes a v_i. O número de arcos que tem um vértice v_i como seu vértice inicial é chamado grau-de-saída do vértice v_i, e o número de arcos que tem v_i como seu vértice final é chamado grau-de-entrada de v_i.

O sucessor de um vértice v_i é todo v_j que seja extremidade final de v_i e o antecessor de um vértice v_i é todo v_j que seja extremidade inicial de um arco que termina em v_i. A figura 4.8 exemplifica os conceitos apresentados.

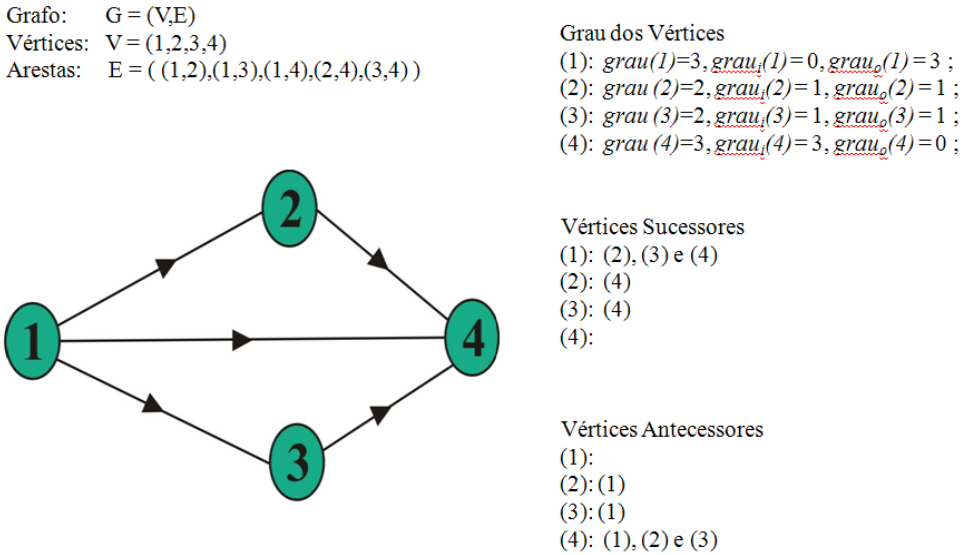


Figura 4.8: Grau, sucessores e antecessores.

Grafo Conexo e Desconexo: Um grafo é denominado conexo quando existe um caminho entre cada par de vértices de *G*, caso contrário é desconexo. Um exemplo dessas definições é ilustrado na figura 4.9. Percebe-se em 4.9(a) que é possível alcançar qualquer vértice do grafo a partir de outro (grafo conexo), o mesmo não pode ser dito para o grafo 4.9(b) onde, por exemplo, a partir do vértice 1 não é possível alcançar o vértice 6 (grafo desconexo).

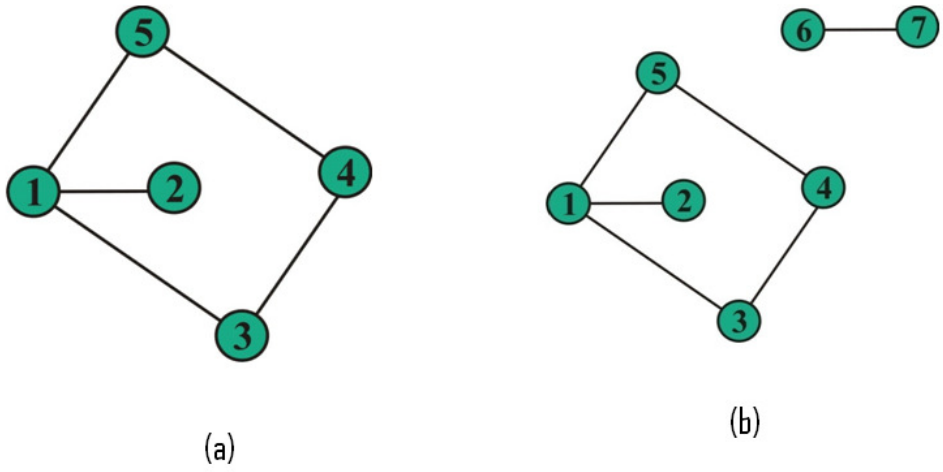


Figura 4.9: (a) Grafo conexo (b) grafo desconexo.

No caso de grafos orientados (dígrafos), um grafo é dito ser fortemente conexo (f-conexo) se todo par de vértices está ligado por pelo menos um caminho em cada sentido. Isto significa que cada vértice pode ser alcançável partindo-se de qualquer outro vértice do grafo (figura 4.10).

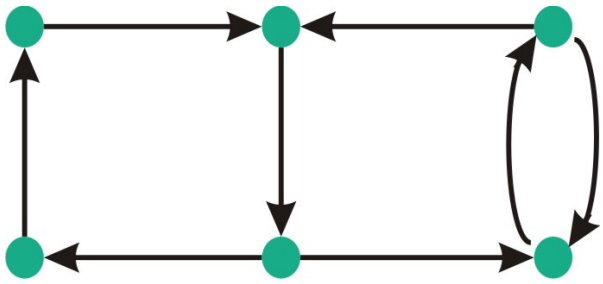


Figura 4.10: Grafo f-conexo.

Base e Anti-Base: Uma base de um grafo $G(V,E)$ é um subconjunto $B \subseteq V$, tal que: dois vértices quaisquer de B não são ligados por nenhum caminho e todo vértice não pertencente a B pode ser atingido por um caminho partindo de B.

Já uma anti-base de um grafo $G(V,E)$ é um subconjunto $A \subseteq V$, tal que dois vértices quaisquer de A não são ligados por nenhum caminho e de todo vértice não pertencente a A pode se atingir A por um caminho. A figura 4.11 apresenta os dois conjuntos, A e B, definidos acima.

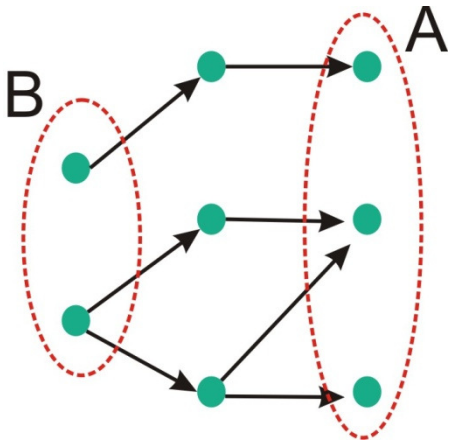


Figura 4.11: Base e anti-base.

Raiz e Anti-Raiz: Se a base de um grafo $G(V,E)$ é um conjunto unitário, então a base é a raiz de G. Se a anti-base de um grafo $G(V,E)$ é um conjunto unitário, então esta anti-base é a anti-raiz de G (figura 4.12).

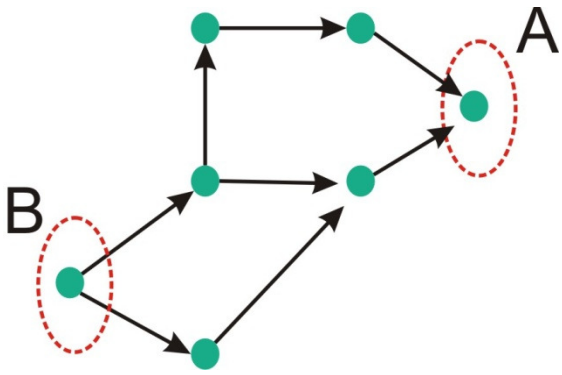


Figura 4.12: Raiz e anti-raiz.

Praticar: Usando o grafo da figura 4.6 responda: ele é um grafo direcional? Possui arestas de laço? Possui arestas paralelas? Qual o antecessor e o sucessor do vértice 5? Qual é o grau de entrada e saída de vértice 3? O grafo é conexo?

2.2. Caminhos e Ciclos

Caminho: Uma seqüência de vértices $v_1, \ldots, v_k$ tal que $(v_j, v_{j+1}) \in E$, $1 < j < |k-1|$ é denominado caminho de v_1 a v_k . Diz-se então que v_1 alcança ou atinge v_k;

Um caminho de k vértices é formado por $k-1$ arestas e o valor de $k-1$ é o comprimento do caminho. Se todos os vértices do caminho $v_1, \ldots, v_k$ forem distintos a seqüência recebe o nome de caminho simples ou elementar e se as arestas forem distintas a seqüência recebe o nome de trajeto.

Considere o grafo da figura 4.13.

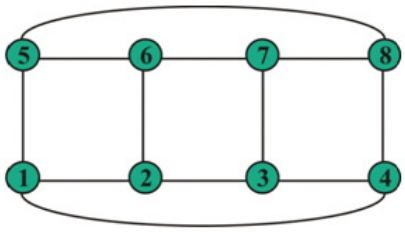


Figura 4.13: Grafo com oito vértices e doze arestas.

Um caminho simples no grafo pode ser 1 – 2 – 3 – 7, já um trajeto seria 2 – 3 – 7 – 6 – 2 – 1 – 5 (figura 4.14). Neste caso, o comprimento do caminho simples é 3 e o comprimento do trajeto é 6.

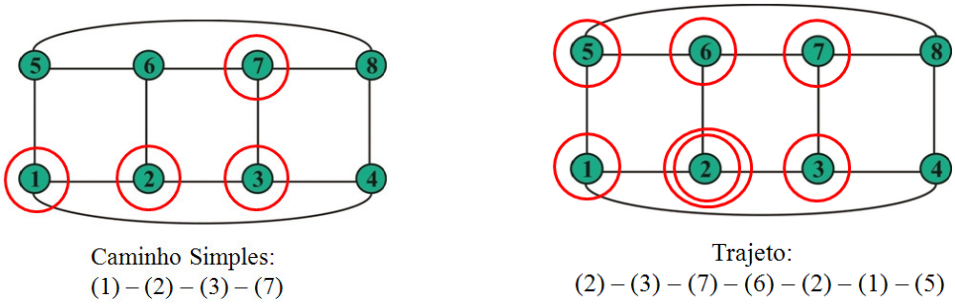


Figura 4.14: Caminho e trajeto em um grafo.

Outro conceito importante relacionado a caminho é a questão da independência. Dois ou mais caminhos são independentes se nenhum deles contém ao menos um vértice interno do outro.

Ciclo: Um ciclo é um caminho $v_1, \ldots, v_k$ sendo $v_1 = v_{k+1}$ e $k \geq 3$. Os grafo que não possuem ciclos são chamados de acíclicos e os que possuem são chamados de cíclicos.

Um ciclo ou caminho que contenha cada vértice do grafo exatamente uma vez é chamado de hamiltoniano e um ciclo ou caminho que contenha cada aresta do grafo exatamente uma vez é chamando de euleriano. A figura 4.15 apresenta dois grafos, um cíclico e outro acíclico e a figura 4.16 apresenta dois caminhos percorridos em um grafo, um hamiltoniano e um euleriano.

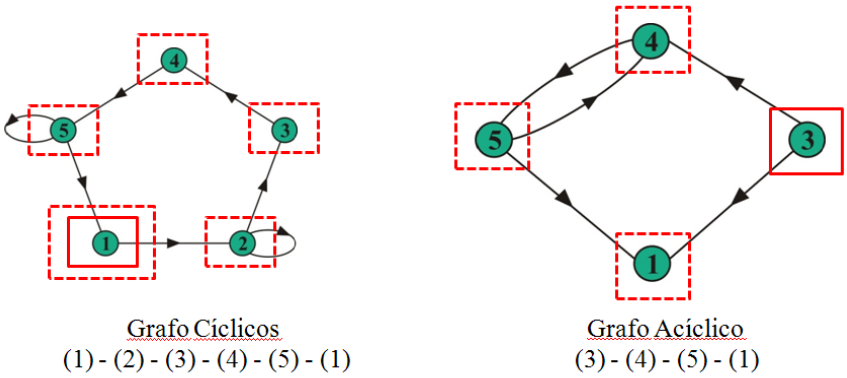
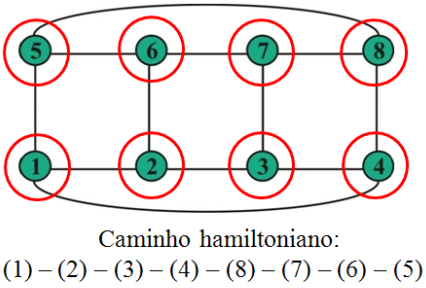


Figura 4.14: Grafos cíclico e acíclico.



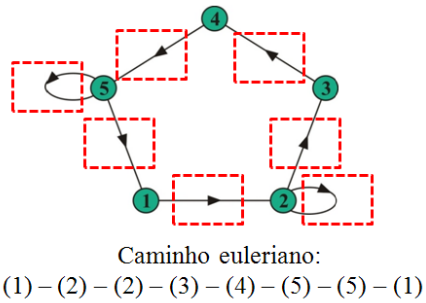


Figura 4.16: Caminho hamiltoniano e caminho euleriano.

Praticar: Usando os grafo da figura 4.16 responda: Em (a) o caminho 2 – 3 – 4 – 8 – 7 – 6 é hamiltoniano? Em (b) o caminho 1 – 2 – 3 – 4 – 5 – 1 é euleriano?

2.3. Tipos de grafos

Grafo Regular: Um grafo é dito ser regular quando todos os seus vértices têm o mesmo grau (figura 4.17).

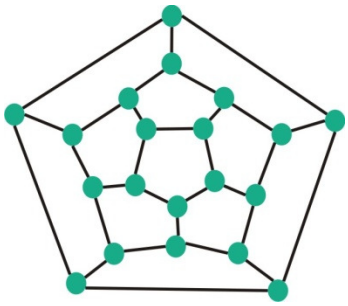


Figura 4.17: Grafo regular.

Grafo Completo: Um grafo é dito ser completo quando há uma aresta entre cada par de seus vértices (figura 4.18). Observe na figura que o índice de *K* representa a quantidade de vértices do grafo e a letra *K* denota que o grafo é completo.

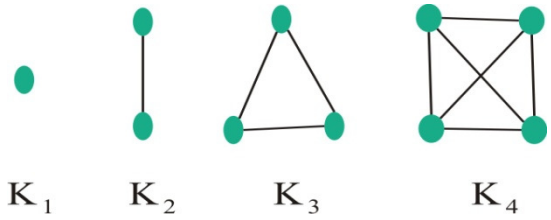


Figura 4.18: Grafo completo.

Grafo Bipartido: Um grafo é dito ser bipartido quando seu conjunto de vértices *V* puder ser particionado em dois subconjuntos V_1 e V_2, tais que toda aresta de *G* une um vértice de V_1 a outro de V_2 (figura 4.19).

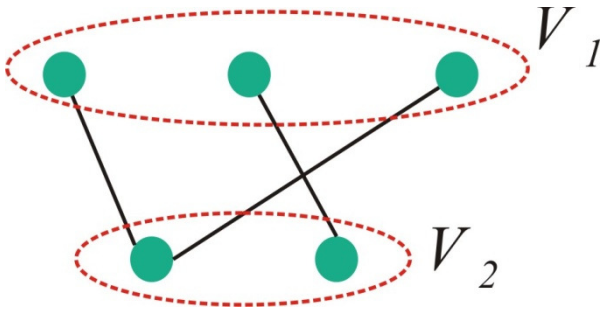


Figura 4.19: Grafo bipartido.

Grafo Valorado: Um grafo $G(V,E)$ é dito ser valorado quando existe uma ou mais funções relacionando V e/ou E com um conjunto de números (figura 4.20).

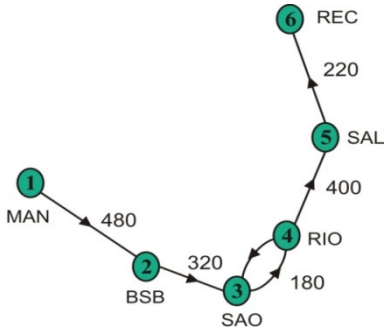


Figura 4.20: Grafo valorado.

Multigrafo: Um grafo $G(V,E)$ é dito ser um multigrafo quando existem múltiplas arestas entre pares de vértices de G (figura 4.21).

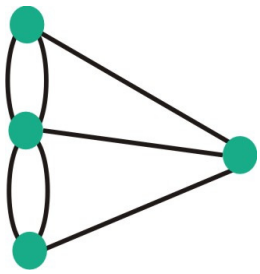


Figura 4.21: Multigrafo.

Subgrafo: Um grafo $G_s(V_s, E_s)$ é dito ser subgrafo de um grafo $G(V,E)$ quando $V_s \subseteq V$ e $E_s \subseteq E$ (figura 4.22).

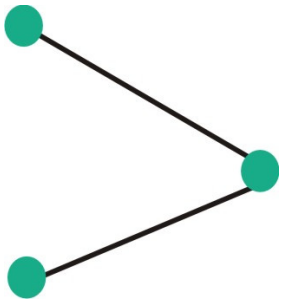


Figura 4.22: Subgrafo.

Praticar: O grafo da figura 4.19 é regular? E o da figura 4.20 é um grafo completo ?

2.4. Complemento de um grafo e Subgrafos

Complemento de um grafo: Denomina-se complemento de um grafo *G(V,E)* ao grafo *G**, o qual possui o mesmo conjunto de vértices que *G* e para todo par de vértices distintos *v, w* $\in$ *V*, tem-se que *(v,w)* é uma aresta de *G** se e somente se não o for de *G.*

Observe na figura 4.23 que as arestas *(1,3), (1,4), (1,5), (2,5) e (3,5)* de *G** não existem em *G* e que *G** possui o mesmo conjunto de vértices *(1,2,3,4,5,6)* de *G*. Logo *G** é o complemento de *G*. Observe também que *G* é um grafo conexo e *G** é um grafo desconexo. Porém, esta característica não viola a definição de complemento de um grafo.

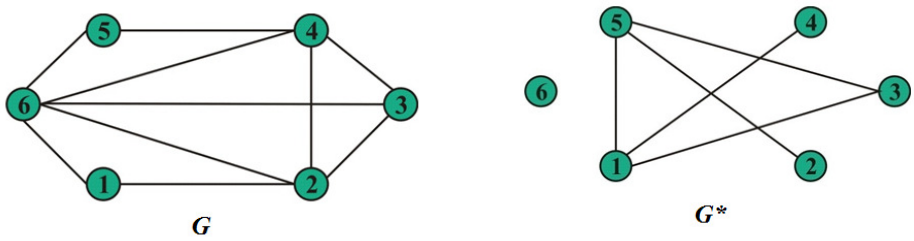


Figura 4.23: Complemento de um grafo.

Subgrafo: Um subgrafo $G_2(V_2,E_2)$ de um grafo $G_1(V_1,E_1)$ é um grafo tal que: $V_2 \subseteq V_1$ *e* $E_2 \subseteq E_1$. A figura 4.24 mostra um grafo *G* e dois subgrafos G_1 e G_2.

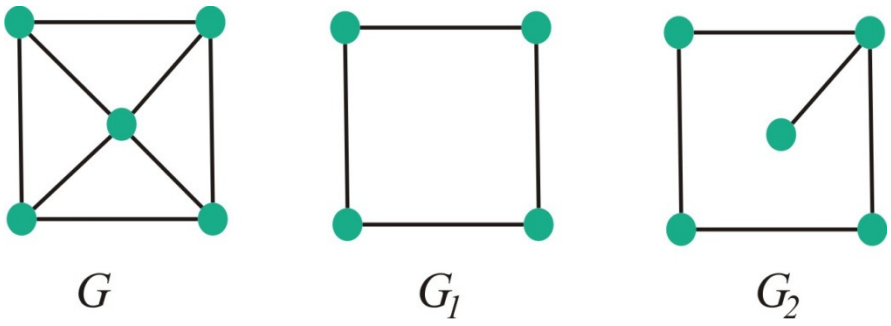


Figura 4.24: Subgrafo.

Observe na figura que o conjunto de vértices de G_1 (1,2,3,4) e o seu conjunto de arestas (*1,2), (2,3), (3,4) e (4,1)* são subconjuntos

do conjunto de vértices e arestas de G, respectivamente. Assim sendo, G_1 é um subgrafo de *G*. De forma análoga pode-se constatar que G_2 também é um subgrafo de *G*.

Se um subgrafo G_n possuir toda aresta *(v,w)* do grafo G_m tal que ambos *v* e *w* estejam em V_n, então G_n é o subgrafo induzido pelo subconjunto de vértices V_n. Diz-se então que V_n induz G_n. A figura 4.25 mostra a partir de um grafo G a obtenção de um subgrafo induzido e de um subgrafo não induzido.

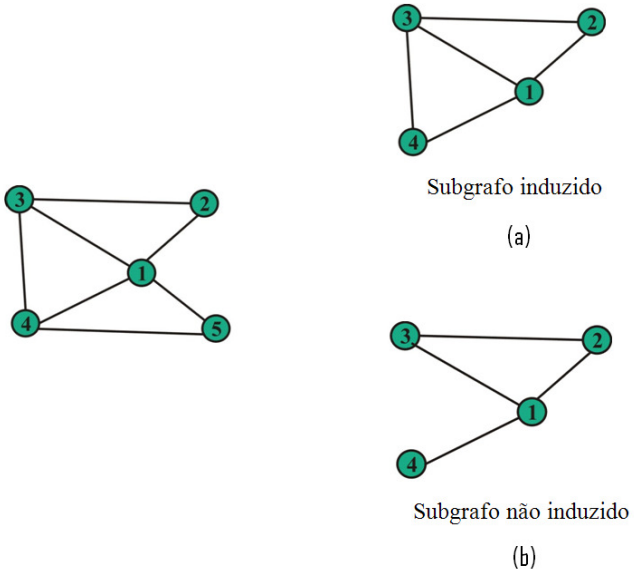


Figura 4.25: Subgrafo induzido e subgrafo não induzido.

Observe que em (a) o subgrafo G_1 obtido a partir de G induzido. O mesmo não pode ser dito para o subgrafo G_2 porque nele não existe a aresta *(3,4) a qual é existente no grafo G.* Por esse motivo pode-se afirmar que G_2 não é um subgrafo induzido de *G*.

Praticar: Utilize os grafos da figura 4.24 e construa o complemento deles. Construa um subgrafo induzido a partir do grafo G da figura 4.23.

2.5. Grafo Hamiltoniano

É um grafo que possui caminho ou ciclo hamiltoniano. Um caminho hamiltoniano é aquele que contém cada nó do grafo exatamente uma vez. A figura 4.26 apresenta um exemplo de grafo hamiltoniano, pois o mesmo contém o ciclo 1, 2, 3, 4, 5, 1.

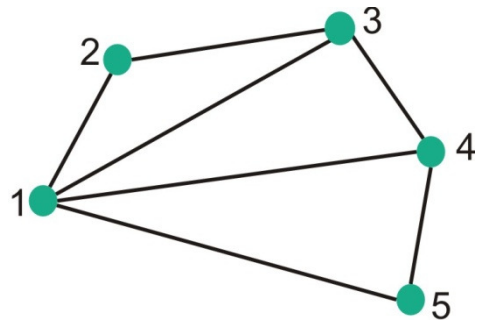


Figura 4.26: Grafo hamiltoniano.

Aplicação — Problema do caixeiro viajante: Neste problema, um caixeiro viajante deseja visitar várias cidades e retornar ao ponto de partida, de forma que a distância percorrida seja a menor possível. Esse problema pode ser modelado através de um grafo ponderado, como mostra o exemplo da figura 4.27.

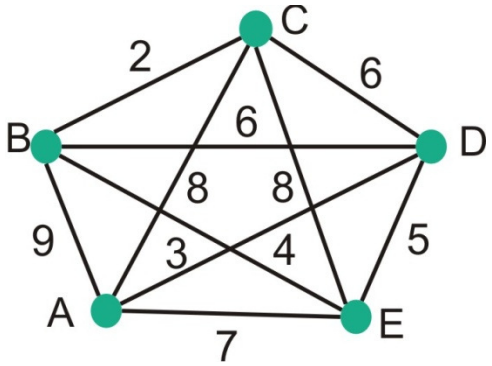


Figura 4.27: Grafo do problema do caixeiro viajante.

Neste caso, deseja-se encontrar um ciclo hamiltoniano de menor peso total. Existem diversos algoritmos que usam heurísticas para resolver rapidamente esse problema, resultando quase sempre na menor distância.

2.6. Grafo Euleriano

É um grafo que possui caminho ou ciclo euleriano. Um caminho euleriano é aquele que contém cada aresta do grafo exatamente uma vez. Caso no caminho encontrado não seja possível retornar ao vértice inicial diz-se que este grafo é então semi-euleriano. Na figura 4.28, G_1 é euleriano (o caminho pode ser a-b-c-d-e-f-a-d-b-e-a), G_2 é semi-euleriano (o caminho pode ser a-e-b-d-c-b-a-d-e) e G_3 não é euleriano, nem semi-euleriano.

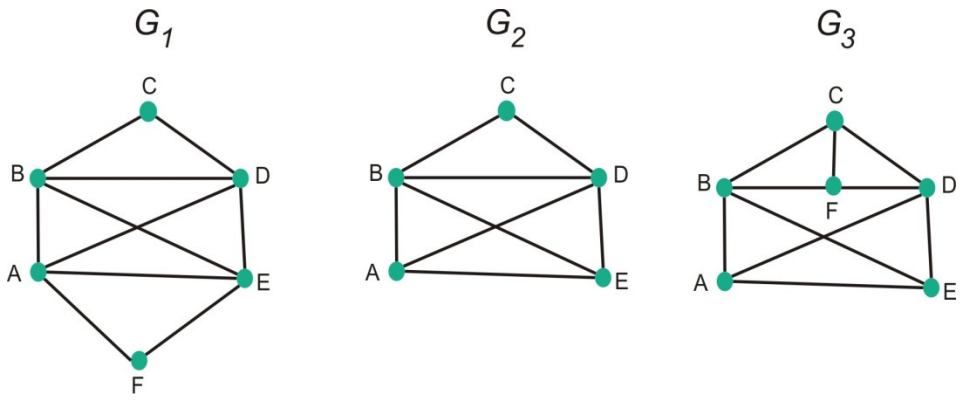


Figura 4.28: Grafo euleriano.

Aplicação — Problema do carteiro chinês: Este problema foi discutido pelo matemático chinês Mei-Ku Kwan. Suponha que um carteiro deseja entregar cartas percorrendo uma menor distância possível e retornar ao ponto de partida. Obviamente ele deve passar por cada estrada em sua rota no mínimo uma vez, mas deve evitar passar pela mesma estrada mais de uma vez. O problema pode ser modelado em termos de um grafo ponderado. Deve-se determinar, portanto, um caminho fechado de peso total mínimo que inclua cada aresta pelo menos uma vez.

Se o grafo é Euleriano, qualquer percurso Euleriano é um caminho fechado, como requerido (e pode ser determinado pelo Algoritmo de Fleury). Caso o grafo não seja Euleriano, o problema é muito mais difícil de ser resolvido (figura 4.29).

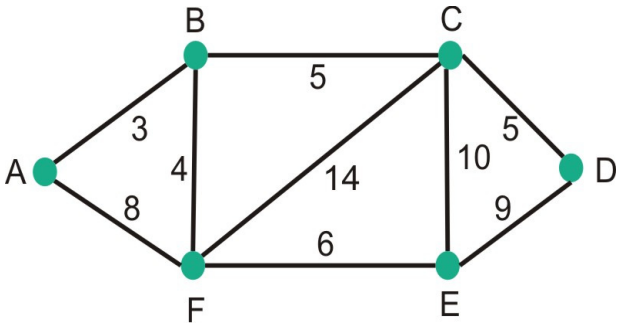


Figura 4.29: Grafo do problema do carteiro chinês.

Praticar: O grafo da figura 4.27 é euleriano? E o da figura 4.29 é hamiltoniano? Justifique sua resposta com a teoria apresentada.

2.7. Planaridade

Um grafo planar é um grafo que admite uma representação gráfica em que as arestas só se encontrem (possivelmente) nos vértices a que são incidentes. Exemplos clássicos de grafos planares é dado pelos grafos que representam os poliedros. Na figura 4.30, apresentamos os grafos dos 5 sólidos platônicos: tetraedro, cubo, octaedro, dodecaedro e icosaedro.

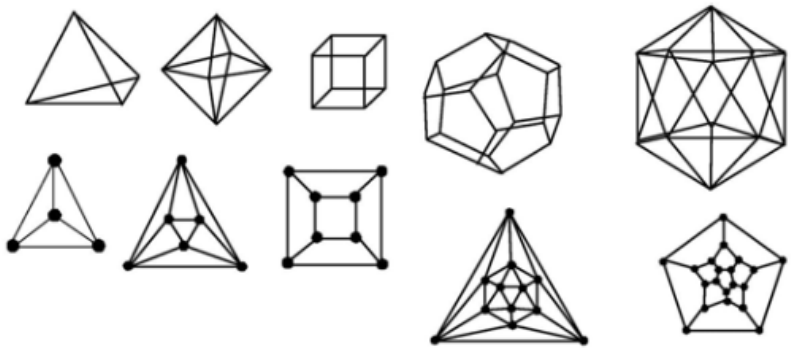


Figura 4.30: Sólidos platônicos.

Em outras palavras, um grafo *G = (V,E)* é planar quando puder ser desenhado em um plano, sem que ocorra cruzamento de arestas, ou seja, duas ou mais arestas não se intersectam

geometricamente exceto nos vértices em que são incidentes. Outros exemplos de grafos planares são dados na figura 4.31.

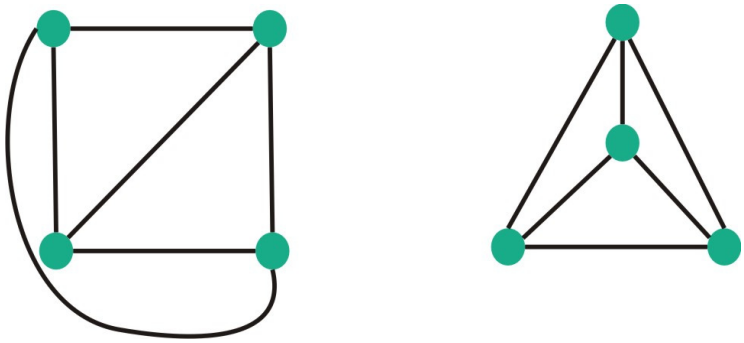


Figura 4.31: Grafo planares.

Seja *G* um grafo planar e *R* uma representação plana de *G* em um plano *P*. Então, as linhas de *R* dividem *P* em regiões, as quais são denominadas faces de *R*. Existe somente uma face não limitada, chamada de face externa ou face infinita. A figura 4.32 ilustra o comentado.

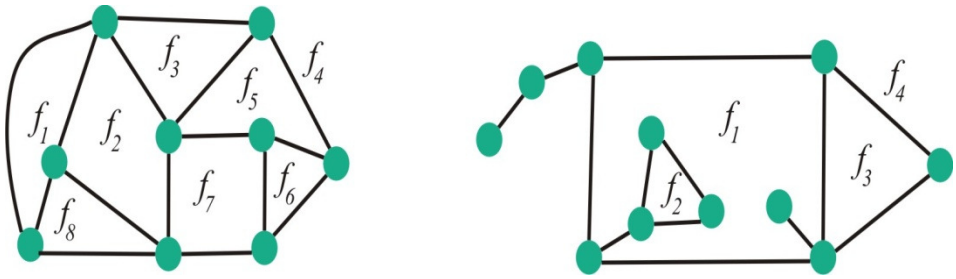


Figura 4.32:Faces em um grafo planar.

Teorema 1 (Euler, 1750) *Seja G um grafo conexo planar e R uma representação plana de G. Seja n, m e f o número de vértices, arestas e faces de G. Então, n – m + f = 2. Esse teorema é conhecido com Fórmula de Euler.*

Uma aplicação muito interessante utilizando os conceitos apresentados de grafos planares pode ser vista nas indústrias eletrônicas de fabricação de circuitos impressos.

2.8. Isomorfismo

Dois grafos G_1 e G_2 são ditos isomórficos se existir uma correspondência um-a-um entre os nós de G_1 para com os nós de G2, tal que o número de arestas unindo quaisquer 2 nós de G_1 é igual ao número de arestas unindo os correspondentes nós de G_2. Um exemplo de grafos isomórficos é dado na figura 4.33.

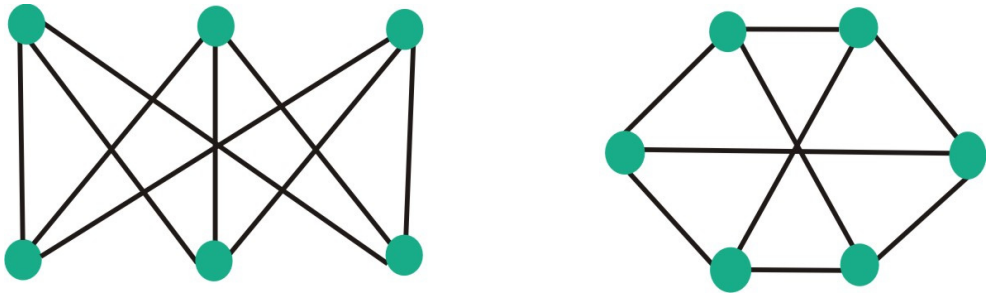


Figura 4.33:Grafos isomorfos.

Na representação de grafos por matrizes de adjacência, que veremos mais adiante, pode-se concluir que dois grafos G_1 e G_2 são isomórficos se, para alguma ordem de seus nós, suas matrizes forem iguais. Uma maneira simples, mas eficiente, de testar o isomorfismo é provar exatamente o contrário através do conceito de invariante. Invariante é uma propriedade que é preservada pelo isomorfismo, como por exemplo, número de nós, grau e determinante da matriz de adjacência.

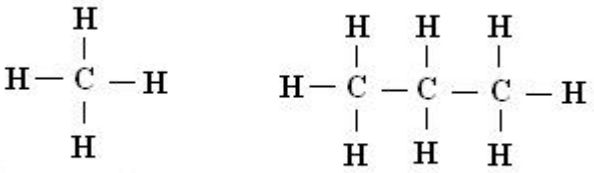
Praticar: Verifique o teorema de Euler para os grafos planares apresentados na figura 4.32. Verifique também se eles são isomorfos.

2.9. Exercícios

1 - A figura abaixo representa as moléculas químicas do metano (CH4) e propano (C3H8).

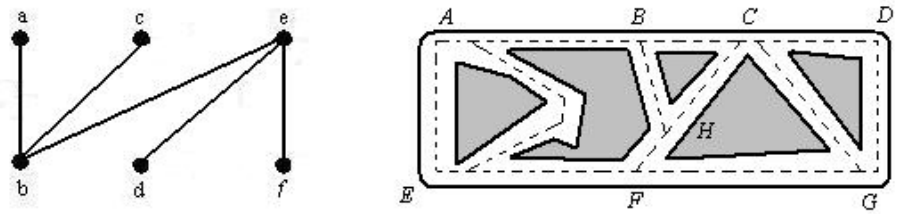
(a) Interpretando esses diagramas como grafos, o que podemos dizer sobre os nós que representam os átomos de carbono (C) e os átomos de hidrogênio (H)?

(b) Existem duas moléculas diferentes com fórmula C4H10. Desenhe os grafos correspondentes a essas moléculas.



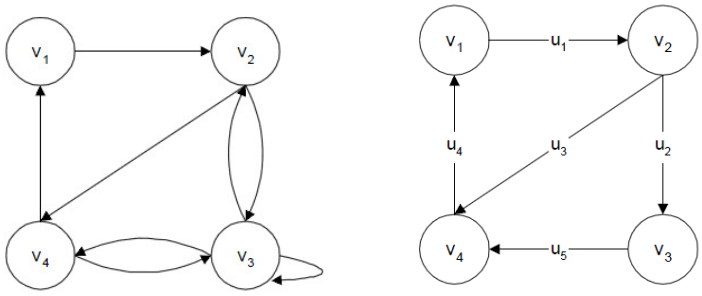
(a) (b)

2 - Para os grafos da figura abaixo escreva o número de nós, arcos e o grau de cada vértice. Para o desenho da (b), desenhe antes sua representação sob forma de grafo.



(a) (b)

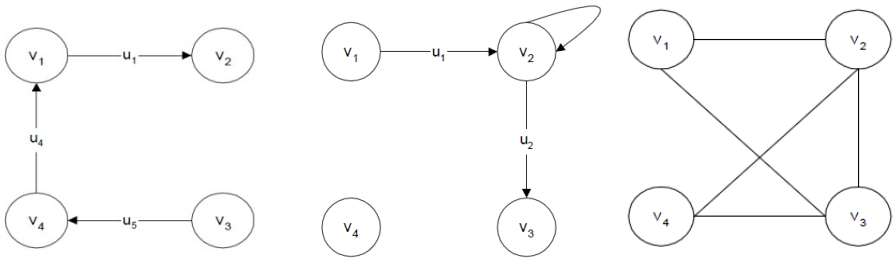
3 - Dados os dígrafos abaixo informe o grau de cada um de seus vértice. Informe também quais são os seus antecessores e seus sucessores.



(a) *(b)*

4 - Desenhe um grafo com 5 vértices (v1, v2, v3, v4, v5), em que grau(v1) = 3, v2 é um vértice ímpar, grau(v3) = 2, e v4 e v5 são adjacentes.

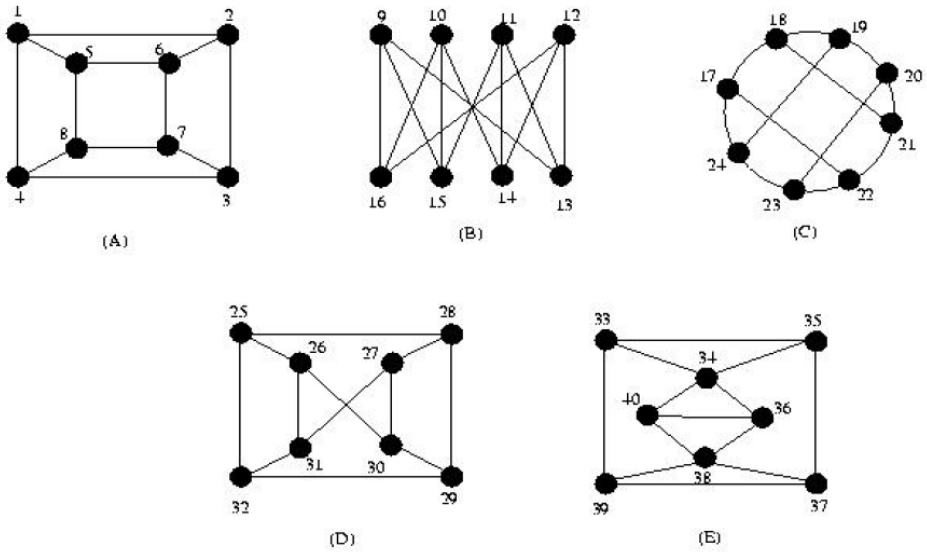
5 - Classifique os grafos abaixo como conexo ou desconexo. Fundamente sua resposta.



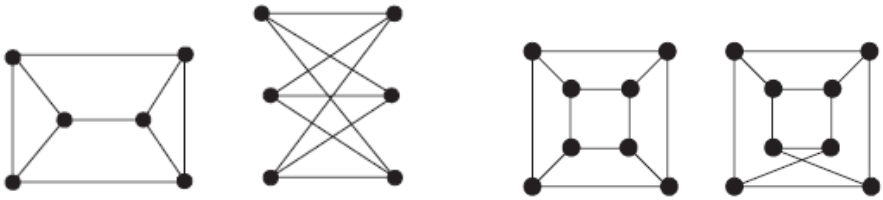
(a) *(b)* *(c)*

6 - Responda sim ou não para cada uma das perguntas abaixo considerando individualmente os grafos abaixo.

a) é regular?
b) é completo?
c) é bipartido?
d) é valorado?
e) é multivalorado?

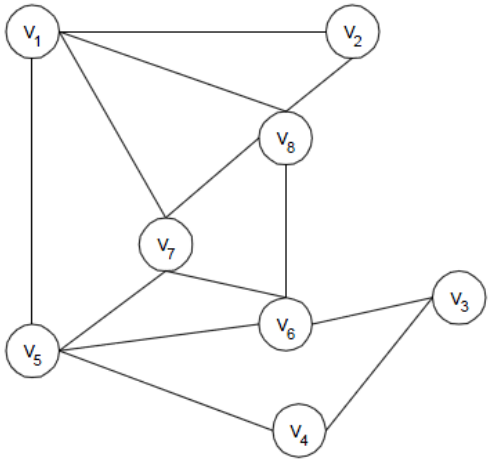


7 - Nos pares de grafos das figuras abaixo mostre qual dos grafos é bipartido e qual não é. Justifique sua resposta.

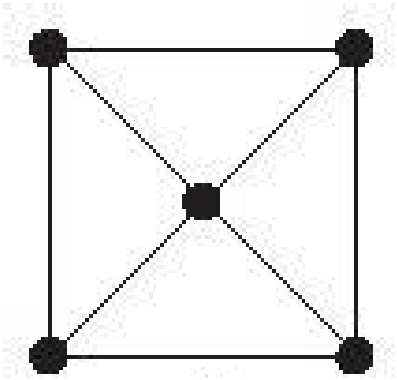


(a) *(b)*

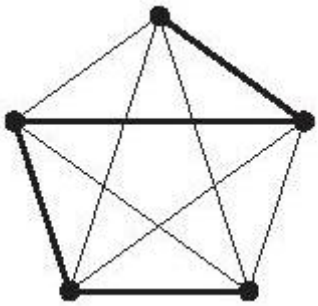
8 - Construa o grafo complementar do grafo G apresentado abaixo.



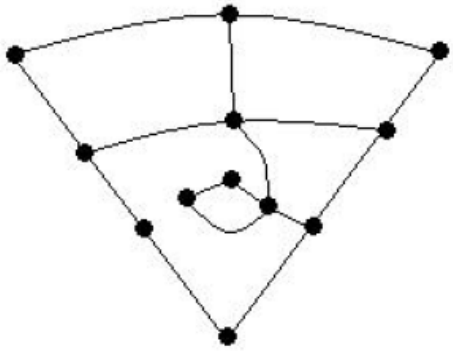
9 - Para o grafo da figura abaixo, desenhe todos os seus subgrafos conexos.



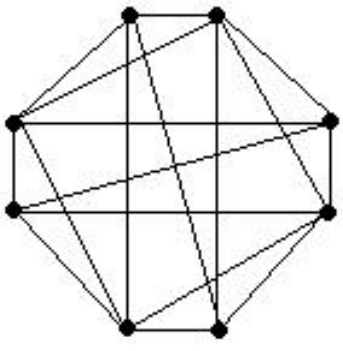
10 - Para o grafo da figura abaixo, desenhe, caso existam, um subgrafo induzido com 3 nós e um subgrafo não induzido com 2 nós.



11 - O grafo da figura abaixo representa um trecho do bairro São Cristóvão, na altura da Av. Dom Severino. Identifique, se houver, um ciclo hamiltoniano e outro euleriano.



12 - Mostre como o grafo abaixo pode ser desenhado em um plano sem cruzamentos, ou seja que o grafo é planar.

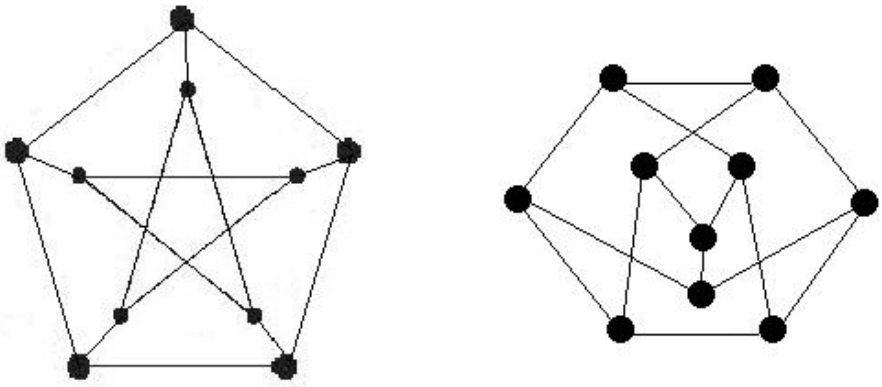


13 - Construa o grafo com seqüência de graus (4, 4, 3, 3, 3, 3):

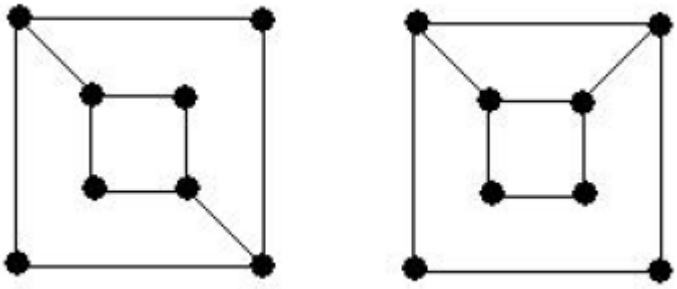
(a) Que seja planar.

(b) Que não seja planar.

14 - Através de uma adequada rotulação dos vértices, mostre que os dois grafos da figura abaixo são isomórficos.



15 - Explique porque os dois grafos da figura abaixo não são isomórficos.



UNIDADE III
INTRODUÇÃO AOS GRAFOS

6. REPRESENTAÇÃO E BUSCA

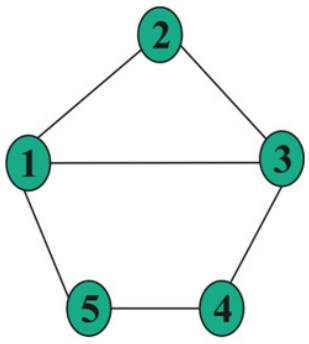
Existem três formas tradicionais de representar os grafos: matriz de adjacência, matriz de incidência e lista de adjacência e todas serão descritas a seguir. Além disso uma operação muito importante em grafos é como percorrê-lo. Apresentaremos aqui duas formas, também tradicionais, de executar essa operação em grafos: busca em largura e busca em profundidade.

3.1. Matriz de Adjacência

Uma das formas mais utilizadas para representar grafos é via a matriz de adjacência. Seja $A = [\ a_{ij}\ ]$ uma matriz $n \times n$, onde n é o numero de vértices de um grafo $G = (V,E)$ qualquer, a matriz de adjacência A é construída da seguinte forma:

$$A(i,j) = \begin{cases} 1 & \text{se } i \sim j \\ 0 & \text{caso contrário} \end{cases}$$

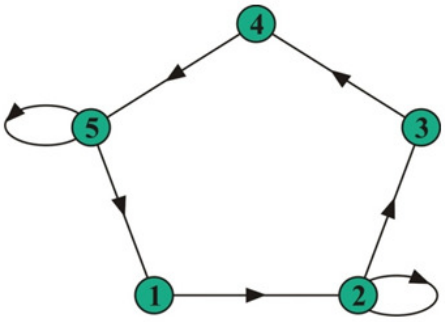
A figura 4.34 ilustra o conceito de matriz de adjacência para um grafo simples.



	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	0	0
3	1	1	0	1	0
4	0	0	1	0	1
5	1	0	0	1	0

Figura 4.34:Matriz de adjacência.

Quando o grafo é ponderado, a representação só fica completa quando também se indica a sua matriz de pesos, construída de maneira semelhante à matriz de adjacência (troca-se o valor do peso pelos 1´s). Para dígrafos, é preciso observar o sentido do caminho entre os nós e adotar um padrão para o sinal dos pesos (figura 4.35).



	1	2	3	4	5
1	0	1	0	0	0
2	0	1	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1
5	1	0	0	0	1

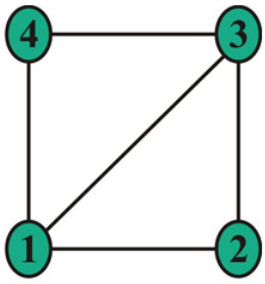
Figura 4.35:Matriz de adjacência para um dígrafo.

3.2. Matriz de Incidência

A matriz de incidência $B = [b_{ij}]$ de um grafo $G = (V,E)$, com $V = (v_1, v_2, \ldots, v_n)$ e $E = (e_1, e_2, \ldots, e_m)$, é definida da seguinte forma:

$$B(i,j) = \begin{cases} 1 & \text{se } v_i \in e_j \\ 0 & \text{caso contrário} \end{cases}$$

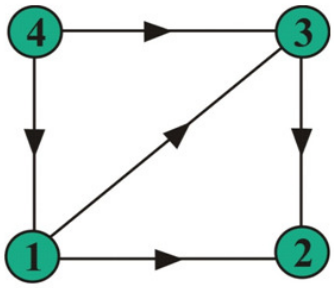
A figura 4.36 ilustra o conceito de matriz de incidência para um grafo simples.



	1,2	1,3	1,4	2,3	3,4
1	1	1	1	0	0
2	1	0	0	1	0
3	0	1	0	1	1
4	0	0	1	0	1

Figura 4.36:Matriz de incidência.

Se *G* é um dígrafo, então $b_{ij} = +1$ se v_i está no início da seta e $b_{ij} = -1$, caso v_i esteja na cabeça da seta. Para grafos ponderados, vale também a mesma observação no que diz respeito à escolha de sinais para representar os arcos e seus pesos (figura 4.37).



	1,2	1,3	3,2	4,3	4,1
1	1	1	0	0	-1
2	-1	0	-1	0	0
3	0	-1	1	-1	0
4	0	0	0	1	1

Figura 4.37:Matriz de incidência para um dígrafo.

3.3. Lista de Adjacência

Trata-se de uma representação que favorece a recuperação mais rápida de informação nos grafos, basicamente por não possuir informações de não adjacência (os zeros na matriz de adjacência). Seja *G(V,E)* um grafo, a estrutura de adjacência *A* de *G* é um conjunto de *n* listas *A(v)*, uma para cada $v \in V$. Cada lista *A(v)* é denominada lista de adjacências do vértice *v*, e contém os vértices *w* adjacentes a *v* em *G*. A figura 4.38 apresenta um exemplo de representação de um grafo por lista de adjacência.

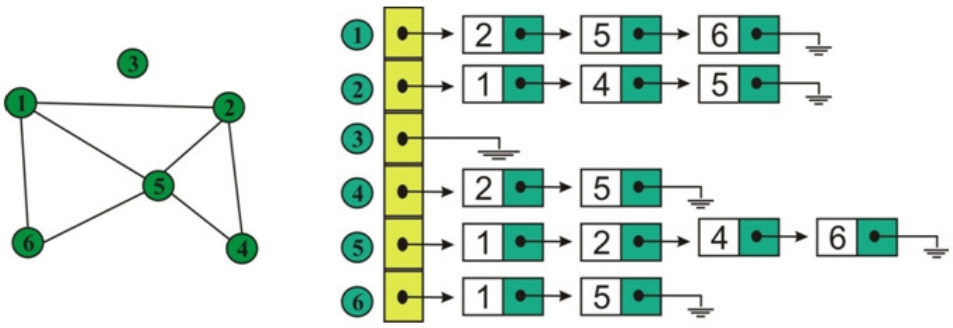


Figura 4.38:Lista de adjacência.

Se o grafo for ponderado, então a informação dos pesos das arestas deve ser armazenada em outra lista em uma correspondência um-a-um com a lista de sucessores de cada vértice.

Praticar: Represente o grafo da figura 4.35 na forma de matriz de incidência, o grafo da figura 4.37 na forma de matriz de adjacência e ambos na forma de listas de adjacências.

3.4. Busca em profundidade

A busca visa resolver uma questão básica: como explorar um grafo? Ou seja, deseja-se obter um processo de como caminhar pelos nós e arestas de um grafo.

Uma busca em profundidade (*Depth-First Search- DFS*) começa em um vértice *v* chamado raiz e caminha por todos os vértices que podem ser alcançados a partir de *v*. Observe que *v* é um parâmetro passado ao algoritmo DFS.

O algoritmo para busca em profundidade marca *v* (raiz) como visitado, pega um vértice *w* conectado a *v* (a aresta *(v, w)* existe) e continua a DFS em *w*. Depois que a busca em *w* termina, o algoritmo toma outro vértice *z* conectado a *v* ainda não visitado e faz a busca em *z*. O algoritmo termina quando todos os vértices ligados a *v* já foram marcados (visitados).

Um pseudo-algoritmo para implementar busca em profundidade em um grafo, cujo elementos dos seus vértices são inteiros, é apresentado a seguir.

Seja um grafo *G = (V,E)* que contém *n* vértices. Seja também uma representação que indica, para cada vértice, se ele foi visitado

ou não. Eis uma versão recursiva do algoritmo de busca em profundidade que visita todos os vértices:

```
procedimento Busca(G: Grafo)
        Para Cada vértice v de G:
                Marque v como não visitado
        Para Cada vértice v de G:
                Se v não foi visitado:
                        Busca-Prof(v)

procedimento Busca-Prof(v: vértice)
        Marque v como visitado
        Para Cada vértice w adjacente a v:
                Se w não foi visitado:
                        Busca-Prof(w)
```

Note que esse algoritmo funciona com um grafo desconexo. Se já sabemos que o grafo é conexo, podemos chamar diretamente a função *Busca-Prof*, escolhendo arbitrariamente um vértice inicial.

Para implementar esse algoritmo, é a lista de adjacência que aparece como estrutura ideal. Isso porque a principal operação efetuada pelo algoritmo é a escolha de um vértice adjacente e já sabemos que nesse caso a estrutura de adjacência é a melhor opção. Supondo então que a estrutura de adjacência é utilizada para implementar a busca, podemos ver que o tempo de execução do algoritmo é em $O(max(a,n))$, onde a e n representam o número de arestas e de vértices, respectivamente. Como cada vértice deve ser visitado, o algoritmo necessariamente fará n chamadas do procedimento *Busca-Prof*, muitas delas recursivas. Em cada chamada, todos os vértices adjacentes serão testados. No total, o número de testes realizados será igual ao número de arestas. Então o tempo total gasto para as chamadas a *Busca-Prof* é em $O(a)$. A esse tempo devemos acrescentar o tempo gasto no procedimento *Busca*: o tempo de inicialização, e o tempos para testar, para cada vértice, se ele foi marcado. No total isso dá um tempo em

O(2n) = O(n). Portanto, a busca em profundidade tem um tempo de execução em *O(a+n).* Na verdade temos um tempos em *O(max(a,n)):* se o grafo tem mais arestas que vértices temos um tempo em *O(a).* No caso contrário, temos um tempo em *O(n).*

Considere agora o grafo ilustrado na figura 4.39 (a) e a estrutura de adjacência ilustrada na figura 4.39 (b) que representa esse grafo.

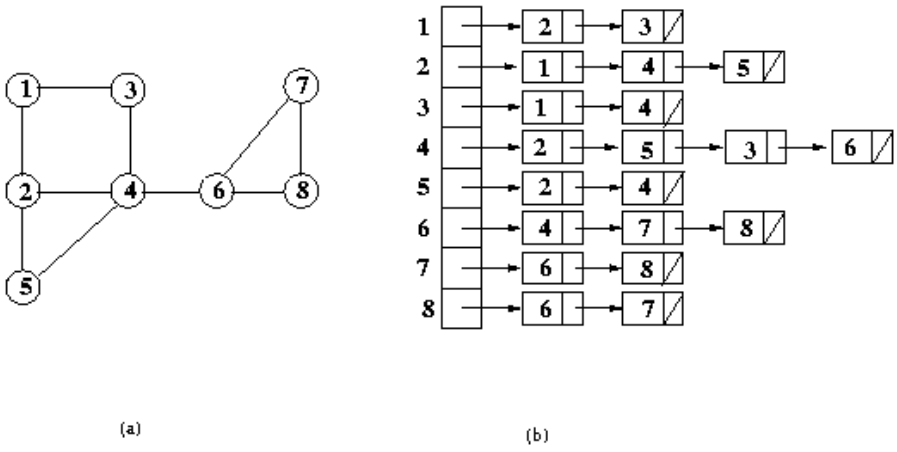


Figura 4.39:Busca em profundidade.

Eis o traço de execução do algoritmo com esse exemplo:

Busca-Prof(1)
Busca-Prof(2)
Busca-Prof(4)
Busca-Prof(5)
Busca-Prof(3)
Busca-Prof(6)
Busca-Prof(7)
Busca-Prof(8)

Quando o algoritmo chega ao vértice 5, não há nenhum vizinho dele que não foi visitado. Ele retorna dessa chamada para

continuar a busca a partir do vértice anterior que é o vértice 4. A busca continua com o próximo vizinho de 4 que não foi visitado: o vértice 3. Como o vértice 3 não tem vizinhos que não foram visitados, o algoritmo retorna imediatamente ao vértice 4, para continuar a busca com o vértice 6, e assim por diante chegamos ao último vértice visitado que é vértice 8.

Observe também que o algoritmo de busca em profundidade pode ser usado para percorrer todos os vértices de um grafo e para descobrir se um grafo é conectado ou não. Se, após uma busca em profundidade, todos os vértices forem visitados, então o grafo é conectado.

3.5. Busca em largura

A idéia da busca em largura consiste em processar todos os nós em um dado nível antes de caminhar para um nível mais alto.

Uma busca em largura (*Breadth-First Search*) é feita em um grafo *G = (V, E)* começando em um vértice v. Primeiro o algoritmo visita *v* e todos os vértices conectados a *v,* chamados filhos de v. Isto é, o algoritmo visita vértices *w* tal que *(v, w)* $\in$ *E*. No segundo passo, o algoritmo visita todos os netos *de v*. Isto é, os vértices que não estão conectados diretamente a *v* mas estão conectados a algum vértice que está conectado a *v*. O algoritmo prossegue deste modo até que todos os vértices alcançáveis por v sejam visitados.

Um pseudo-algoritmo para implementar busca em profundidade utilizando árvore geradora é apresentado a seguir.

Contrariamente à busca em profundidade, o algoritmo de busca em largura não é recursivo. Mas pode ser comparado à versão iterativa da busca em profundidade, que usa uma pilha P:

```
procedimento Busca-Prof-Iter(v: vértice)
        Inicializar P
        Marcar v como visitado
        Empilhar v em P
        Enquanto P não vazio:
                Enquanto existe um vértice w não visitado e adjacente ao
                vértice no topo de P:
                        Marcar w como visitado
                        Empilhar w em P
                Retirar o primeiro elemento de P
```

O algoritmo de busca em largura é semelhante ao algoritmo de busca em profundidade. A principal diferença é que usamos um fila *F* ao invés de uma filha:

```
procedimento Busca-Largura(v: vértice)
        Inicializar F
        Marcar v como visitado
        Colocar v no final de F
        Enquanto F não vazio:
                u := primeiro elemento de F
                Retirar u de F
                Para cada vértice w adjacente a u:
                        Se w não foi visitado:
                                Marcar w como visitado
                                Colocar w no final de F
```

Nos dois casos é preciso um programa para lançar a busca:

```
procedimento Busca(G: Grafo)
        Para cada vértice v de G:
                Marcar v como não visitado
        Para cada vértice v de G:
                Se v não foi visitado:
                        {Busca-Largura ou Busca-Prof-iter}(v)
```

Praticar: Implemente, em Java, os algoritmos de busca e largura e busca em profundidade apresentados. Pesquise outras formas de implementar busca em largura e busca em profundidade sem utilizar recursividade.

3.6. Exercícios

1 – Monte um grafo a partir da matriz de adjacência (a) e outro a partir da matriz de incidência (b).

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(a)

	1	2	3	4	5	6	7	8
1	1	1	0	0	0	0	0	0
2	0	-1	-1	1	0	0	0	0
3	0	0	0	0	0	1	1	0
4	-1	0	1	0	-1	0	0	0
5	0	0	0	-1	1	-1	0	0
6	0	0	0	0	0	0	-1	0

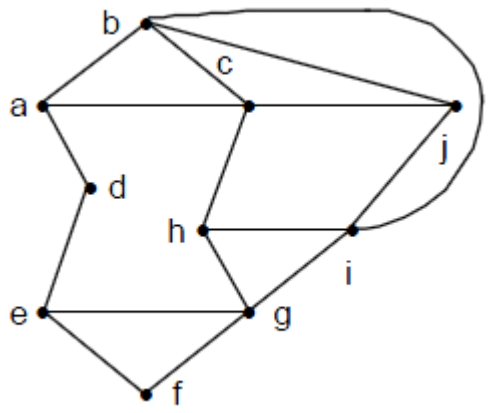
(b)

2 - Dado o grafo da figura abaixo, determine:

(a) matriz de adjacência;

(b) matriz de incidência;

(c) lista de adjacência.

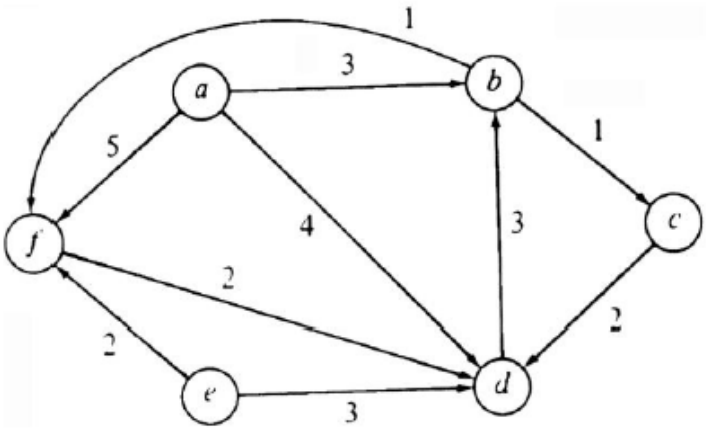


3 - Dado o dígrafo ponderado da figura, determine:

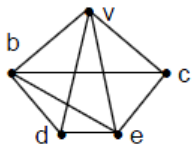
(c) matriz de adjacência;

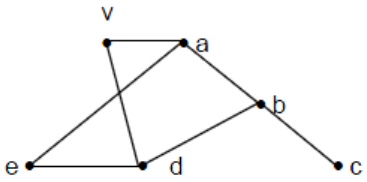
(d) matriz de incidência;

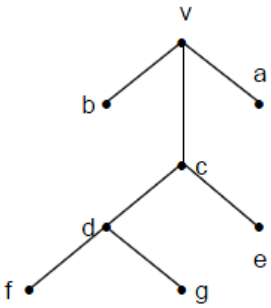
(c) lista de adjacência.



4 – Represente os grafos abaixo na forma de matriz de adjacência, matriz de incidência, lista de adjacência. Faça neles uma busca em profundidade e uma busca em largura começando no vértice v.







5 – Faça uma busca em largura, a partir do vértice 0, no grafo definido pelo conjunto de arestas abaixo:

8-9 3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4 .

5 – Suponha que o grafo da questão anterior está representado por sua matriz de adjacência. Represente-o utilizando matriz de incidência e lista de adjacência.

6 – Represente o grafo abaixo por listas de adjacência. Insira as arestas, na ordem dada, num grafo inicialmente vazio.

8-9 3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

7 – Escreva uma função que use busca em largura para calcular o número de componentes (vértices e arestas) de um grafo.

8 – Implemente a busca em profundidade para o grafo da questão 2. Inicie a busca a partir do nó a. Apresente o desenvolvimento do algoritmo de busca.

9 – Para o mesmo grafo da questão 2, implemente a busca em largura partindo do nó a. Apresente o desenvolvimento.

10- Dado o grafo da questão 3, implemente a busca em profundidade e em largura. Escolha o nó inicial.

7. WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI
www.ufpi.br/uapi

Universidade Aberta do Brasil – UAB
www.uab.gov.br

Secretaria de Educação à Distância do MEC - SEED
www.seed.mec.gov.br

Associação Brasileira de Educação à Distância – ABED
www.abed.org.br

Uma Introdução Sucinta à Teoria dos Grafos
http://www.ime.usp.br/~pf/teoriadosgrafos/

Teoria dos Grafos
http://www.inf.ufsc.br/grafos/

Teoria dos Grafos
http://www.icmc.usp.br/manuals/sce183/grafos.html

Teoria dos Grafos, na Wikipédia:
http://pt.wikipedia.org/wiki/Teoria_dos_grafos

Algoritmos para grafos
http://www.ime.usp.br/~pf/algoritmos_para_grafos/

Algoritmos em grafos
http://www.ime.usp.br/~pf/algoritmos_em_grafos/index.html
Open Problems – Graph Theory and Combinatorics, de Douglas West:
http://www.math.uiuc.edu/~west/openp/
Algoritmos Animados em Java
http://gdias.artinova.pt/projecto/pt/menu_applets.php

Applet Busca em Largura
http://www.cs.duke.edu/csed/jawaa/BFSanim.html

Applet Busca em Profundidade
http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic26/#java

Applet Busca em Profundidade grafo direcionado com pilha
http://www.cs.duke.edu/csed/jawaa/DFSnew.html

8. REFERÊNCIAS BIBLIOGRÁFICAS

BALAKRISHNAN, J. & RANGANATHAN, K. , ***A Textbook of Graph Theory***, Ed. Springer-Verlag,1999.

BOAVENTURA, P. O. , ***Grafos: Teoria, Modelos, Algoritmos***, Ed. Edgard Blucher, 1996.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C. , ***Algoritmos***, Ed. Campus, 2002.

DIESTEL, R. ***, Graph Theory***. Ed. Springer, 1997.

DROZDEK, A. , ***Estruturas de Dados e Algoritmos em C++***, Ed. Thomson, 2002.

FUIRTADO, A. L. , ***Teoria dos Grafos: Algoritmos***, Ed. LTC, 1973.

GOODRICH, M. T. & TAMASSIA, R. , ***Estruturas de Dados e Algoritmos em Java***, Ed. Bookman, 2007.

GOULD, R. , ***Graph Theory***. The Benjamim/Cummings Publishing Company, 1988.

LAFORE, R. , ***Estruturas de Dados e Algoritmos em Java***, Ed. Ciência Moderna, 2004.

LAUREANO, M. ***, Estrutura de Dados com Algoritmos e C***, Ed. Brasport, 2008.

LORENZI , F. & MATTOS , P. N. & CARVALHO, T. , ***Estruturas de Dados***, Ed. Cengage Learning, 2006.

PEREIRA***, Estruturas de Dados Fundamentais: Conceitos e Aplicações***, Ed. Érica, 2006.

PREISS, B. R. , ***Estruturas de Dados e Algoritmos***, Ed. Campus, 2000.

PUGA, S. & RISSETI, G. , ***Lógica de Programação e Estruturas de Dados***, Ed. Prentice-Hall, 2004.

RANGEL, J. L. & CERQUEIRA, R. & CELES, W. , ***Introdução a Estrutura de Dados***, Ed. Campus, 2004.

SCHILDT, H. , ***C Completo e Total***, Ed. Makron Books, 1996.

SILVA, O. Q. , ***Estruturas de Dados e Algoritmos Usando C: Fundamentos e Aplicações***, Ed. Ciência Moderna, 2007.

SZWARCFITER, J. L. & MARKENZON, L. , ***Estruturas de Dados e Seus Algoritmos***, Ed. LTC, 1994.

TENEMBAUM, A. M. ***, Estruturas de Dados Usando C***, Ed. Makron Books, 1995.

VELOSO, P. A. S. , ***Estruturas de Dados***. Ed Campus, 1983.

WEST, D. , ***Introduction to Graph Theory***, Prentice Hall (1996)

WILSON, R. J. , ***Introduction to Graph Theory***, Ed. AddisonWesley, 1996.

WIRTH, N. , ***Algoritmos e Estruturas de Dados***, Ed. LTC, 1989.

ZIVIANI, N. . ***Projeto de Algoritmos com implementação em PASCAL e C***, Ed. Thomson, 2005.

AUTORES

André Macêdo Santana

CV. http://lattes.cnpq.br/5971556358191272

Mestre em Engenharia Elétrica pela Universidade Federal do Rio Grande do Norte UFRN, Natal RN Brasil (2007). Graduado em Ciência da Computação pela Universidade Federal do Piauí UFPI, Teresina PI Brasil (2004). Atualmente é Professor Assistente do Departamento de Informática e Estatística DIE da UFPI e aluno de doutorado do Programa de Pós-Graduação em Engenharia Elétrica e Computação PPGEEC da UFRN. Atua principalmente nos seguintes temas: Robótica Móvel, Visão Computacional, Filtragem Estocástica, Inteligência Artificial.

AUTORES

Erico Meneses Leão

CV. http://lattes.cnpq.br/1117954290627743

Possui graduação em Bacharelado em Ciência da Computação pela Universidade Federal do Piauí, Brasil, em 2005, e Mestre em Ciências pelo Programa de Pós-Graduação em Engenharia Elétrica, com ênfase em Engenharia de Computação, da Universidade Federal do Rio Grande do Norte, Brasil, em 2007. Foi professor do curso de Ciência da Computação do Centro de Ensino Unificado de Teresina - CEUT e do curso de Sistema de Informação da Faculdade de Atividades Empresariais de Teresina - FAETE. Desde 2008, ele é professor do quadro efetivo Assistente do Departamento de Informática e Estatística - DIE da Universidade Federal do Piauí - UFPI. Seus principais interesses de pesquisa e atuação incluem arquitetura de sistemas distribuídos de tempo real, sistemas operacionais, redes de computadores e sistemas de comunicação de tempo real para redes industriais.