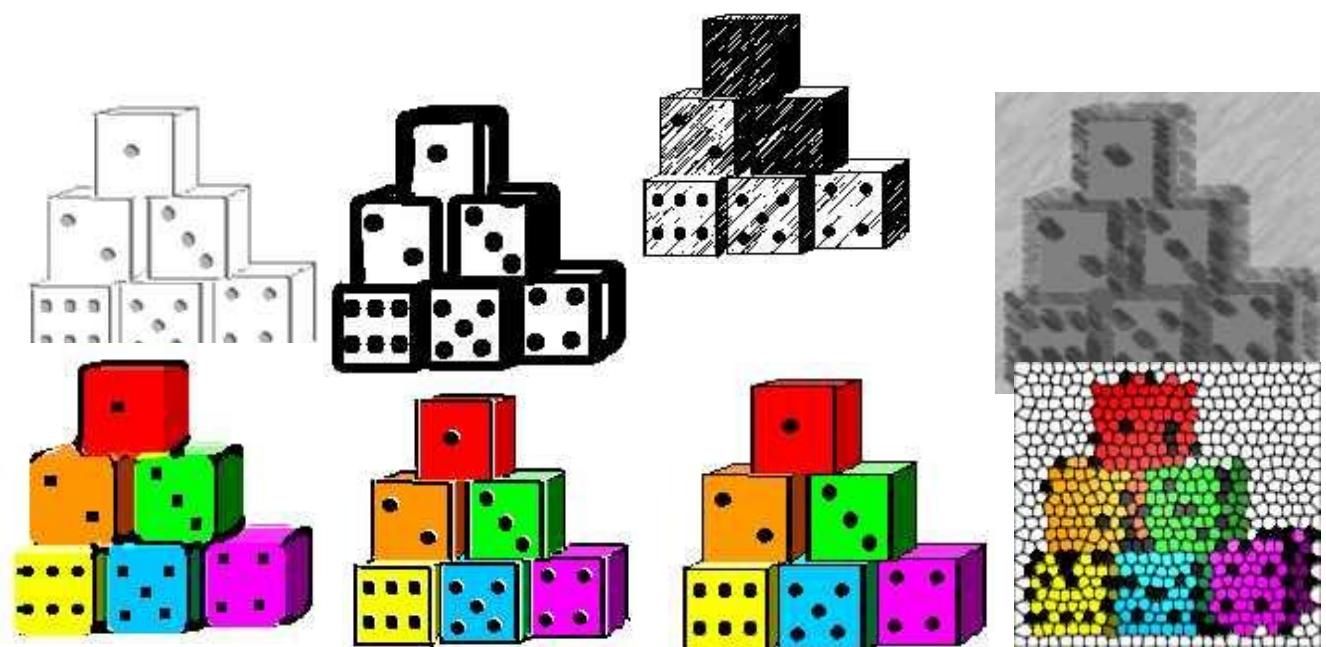


COTUCA

Departamento de Processamento de Dados

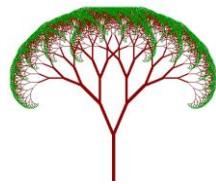


Estruturas de Dados



ÍNDICE

1. Estruturas de Dados e Orientação a Objetos	3
2. Ponteiros – A força	6
3. Listas ligadas	8
4. Hashing	36
5. Filas	64
6. Pilhas	75
7. Programação Recursiva	111
8. Árvores	139
9. Grafos	205
10. Ordenação	238
Apêndice	
1. Framework .Net	267
2. Linguagem C#	274
3. Namespaces e Métodos	287
4. Objetos e Classes	293
5. Gráficos no C#	314
6. C# e a Interface IComparable	321
7. Listas ligadas revisitadas	331
8. Algumas Estruturas em Delphi e Java	339
9. Análise de Algoritmos	367



BIBLIOGRAFIA

- Data Structures and Algorithms with Object-Oriented Design Patterns in C#** - Bruno Preiss
- Data Structures and Algorithms using C#** - Michael McMillan
- Estruturas de Dados e Algoritmos em Java** – Michael Goodrich e Roberto Tamassia – Bookman
- Algoritmos e Estruturas de Dados com Delphi** – Julian Bucknall – Berkeley
- Data Structures Using Pascal** – Tanenbaum
- Algorithms+Data Structures = Programs**, Niklaus Wirth, ed. McGraw-Hill
- Estruturas de Dados usando C** – Tanenbaum – Makron Books
- Programação Estruturada com Estudo de Casos em Pascal**, William J. Collins
- Estruturas de Dados**, Paulo Veloso e outros
- Data Structures, Data Abstraction: A contemporary study using C++** - Mitchell L Model., Prentice Hall International Editions

1. Estruturas de Dados

O estudo de Estruturas de Dados é uma parte muito importante do estudo mais geral de algoritmos. Segundo Niklaus Wirth, criador da linguagem Pascal, um programa é resultante da "soma" entre algoritmos e estruturas de dados. Do mesmo modo que números podem ser representados em sistemas diferentes, dados podem ser armazenados em estruturas diferentes. Em outras palavras, o programador pode se utilizar de diversos métodos para armazenar os dados na memória e no disco para serem processados.

Essas estruturas podem ser escolhidas dentre vários tipos, e cada escolha vai afetar o algoritmo do problema que se pretende resolver, tanto em termos de memória gasta como em tempo de execução dispendido.

Assim sendo, estudar Estruturas de Dados consiste na análise das diversas formas de armazenar os dados, determinação da eficiência de cada tipo de estrutura e escolha do algoritmo mais adequado para cada problema.

Se existe um conjunto de dados de tamanho N para ser processado, a escolha de uma estrutura de dados A para armazenar este conjunto fará que a memória gasta possa variar de um fator de 1 a muitas vezes a memória que seria necessária com outra estrutura B. O tempo de execução, por sua vez, pode variar de 1 a mais de N vezes o tempo gasto por outra estrutura. Portanto, a escolha da estrutura de dados deve ser feita de acordo com a complexidade do problema a resolver, com a memória disponível e o tempo que se pode gastar para resolver o problema. Deve-se aprender a determinar a estrutura de dados adequada para cada tipo de problema.

As operações básicas que são feitas com os dados são: inclusão de um dado novo, procura, exclusão e alteração de um dado já existente. Para que se possa fazer estas operações, às vezes é necessário ordenar os dados, ou guardá-los em estruturas auxiliares para uso posterior.

Para tanto, nosso estudo de estruturas de dados englobará a definição de cada estrutura, bem como métodos (algoritmos) para se operar com elas.

As estruturas que veremos podem ser divididas em dois grupos, quanto à sua disposição na memória: as estruturas de alocação estática, e as estruturas de alocação dinâmica.

Quanto aos algoritmos, podem ser divididos em iterativos e recursivos. Para fazê-los, usaremos, geralmente, a linguagem C#.

Para representar as estruturas, é interessante deixar de lado, num primeiro momento, as declarações específicas das várias linguagens de programação, e procurar-se definições abstratas, ou seja, que não levem em consideração detalhes de implementação. Dessa maneira, pode-se usar os conceitos de estruturas de dados independentemente de implementações em linguagens específicas. Em outras palavras, os detalhes de implementação não devem criar uma dependência no programa; dessa forma, o programa apenas invocará as rotinas que armazenam ou recuperam os dados, sem se preocupar com os detalhes internos dessas rotinas. Dentro delas, o programador utilizará uma implementação específica para a estrutura escolhida.

Para o programador de uma aplicação, pode ser indiferente se a estrutura onde armazena seus dados é uma árvore binária ou um vetor. Cada uma dessas estruturas possui suas peculiaridades, suas limitações e facilidades de uso, mas deve-se procurar separar os programas aplicativos da complexidade de cada implementação específica de uma estrutura de dados. O aplicativo apenas invoca a rotina, passando e recebendo determinados parâmetros, e o programa não se preocupa com a estrutura em si. Uma mesma aplicação, feita por diferentes programadores, pode usar as mesmas chamadas de rotinas, mas usando implementações diferentes dessas rotinas. Assim, o

aplicativo A utiliza rotinas que implementam as estruturas como listas ligadas, enquanto o programa B usa rotinas que implementam árvores. As rotinas chamadas serão, para A e B, por exemplo, Incluir(Elemento), Excluir(Elemento), Pesquisa(Elemento, Onde). A implementação específica de cada biblioteca de rotinas é que mudará, mas não a interface entre as rotinas e os programas que as utilizam.

Como analogia, quando você usa um número real em Clipper ou COBOL, ele tem uma forma de armazenamento diferente de um número real em Pascal ou Java; no entanto, você não precisa saber como ele é armazenado fisicamente. Basta utilizá-lo, e os detalhes de acesso e representação ficam a cargo das rotinas que manipulam reais, em cada uma dessas linguagens.

Alocação Estática de Memória

Alocação é uma palavra que significa reserva de lugar. As variáveis globais de um programa possuem o que se chama de alocação estática de memória, pois uma vez tendo sido declaradas não podem mais ser destruídas ou ter seu tamanho modificado. Como exemplo de estruturas de alocação estática de memória, temos os vetores, strings e matrizes. Essas estruturas de dados, depois de declaradas, já passam a existir desde o início do programa principal e não podem ter seus tamanhos modificados depois de criados.

Várias operações podem ser realizadas com esse tipo de estrutura, algumas das quais já vimos no semestre anterior.

Alocação Dinâmica de Memória

Uma variável é criada dinamicamente quando necessária e descartada quando não mais é usada pelo programa.

Quando a aplicação precisa dessa variável, solicita ao sistema operacional que a crie, e um espaço na memória é reservado para ela, um indicador do local onde ela se encontra é devolvido ao programa, e este pode usá-la conforme necessário.

Essas variáveis não são criadas durante a compilação do programa, como ocorre com as variáveis estáticas globais; pelo contrário, seu espaço de memória é alocado quando o programa precisa.

Exemplos: listas ligadas e árvores binárias.

Nesse tipo de alocação, a reserva de **áreas de memória** é feita em **tempo de execução** de programas, ou seja, de acordo com a evolução da necessidade de espaço para os dados **durante** seu processamento, a memória necessária para armazená-los vai sendo requisitada (**alocada**) ou **liberada** quando não mais é preciso.

A memória de um computador é dividida em duas partes principais: **STACK** e **HEAP**. Stack é a área onde são armazenadas as variáveis estáticas e o programa em execução. HEAP é a área onde se reserva espaço para variáveis criadas com alocação dinâmica.

A alocação dinâmica permite formas de tratamento dos dados que não são facilmente implementadas com vetores, matrizes e pilhas, como inclusão e exclusão de elementos no interior da estrutura de armazenamento, **sem que seja necessário deslocar elementos**, o que, como vimos, é ineficiente.

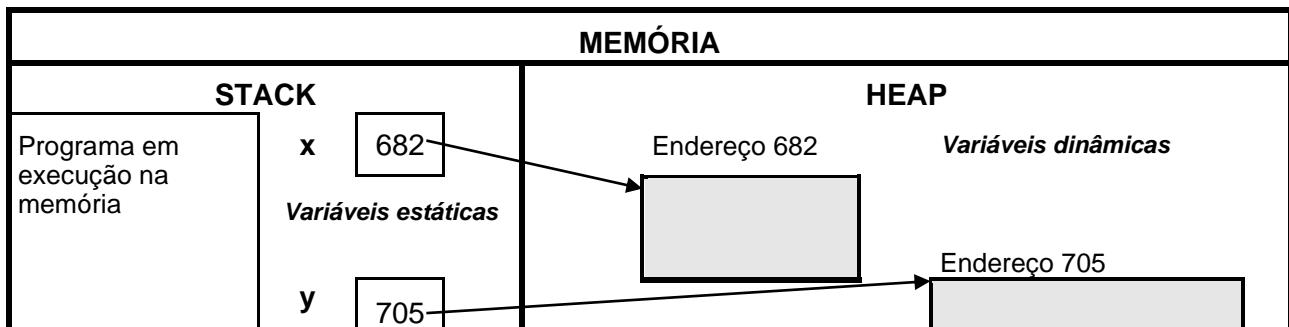
A alocação dinâmica permite ao programador **gerenciar melhor o gasto de memória** pelo programa. Dessa maneira, não é preciso usar mais memória do que o estritamente necessário para armazenar o conjunto de dados processado no momento. O programa se encarregará de requerer mais espaço quando for preciso, de maneira que não é necessária manutenção do programa a cada aumento do número de registros a processar, como ocorreria com um vetor.

Além de, em média, gastar-se menos memória (já que a memória usada é a necessária para armazenar o conjunto dos dados), ganha-se em tempo de execução nas operações de inclusão e exclusão. No entanto, rotinas de pesquisa podem ser mais lentas do que em estruturas de alocação estática, uma vez que **o acesso** a cada elemento da estrutura de **alocação dinâmica**

não é direto, como ocorre num vetor, e sim por meio **indireto**, pelo uso de **variáveis apontadoras**, que **contêm o endereço dos elementos**.

Apontadores são, portanto, **variáveis que armazenam endereços de outras variáveis** que, por sua vez, contêm os dados a serem processados. A alocação destas variáveis com os dados pode ser ou não dinâmica. No entanto, em geral a alocação das variáveis que são acessadas por meio de apontadores é dinâmica.

Os apontadores são necessários para armazenar o endereço de uma **variável dinâmica** porque **não se conhece o endereço dessa variável até o momento em que ela é criada**. Assim, quando o programa requisita a criação de uma variável, o gerenciador memória verifica o tamanho ocupado pela mesma, e procura uma região do HEAP onde ela caiba. O **endereço inicial dessa posição** é então **armazenado na variável apontador**, de modo que este passa a ser o único meio de acesso à variável dinâmica recém-criada, como ilustra a figura 14 abaixo:



- (1) O programa solicita a reserva do espaço e informa qual variável receberá o apontador
- (2) O gerenciador de memória aloca o espaço necessário para a variável dinâmica e devolve o endereço da região alocada
- (3) A variável apontadora indicada em (1) (x e y, na figura) recebe o endereço de memória determinado em (2) e, assim, passa a apontar para a região alocada na memória dinâmica

Figura 1 – esquema de alocação dinâmica de memória

Como exemplo de estruturas de dados implementadas com alocação dinâmica, estudaremos as **Listas Ligadas e Árvores**, além de retomarmos o estudo de Pilhas e Filas, implementando-as com alocação dinâmica ao invés de usar vetores. Todas essas estruturas podem ser implementadas com o uso de vetores, ou seja, com alocação estática, mas aprenderemos as vantagens da alocação dinâmica. Para o estudo de árvores, antes aprenderemos técnicas de programação usando **algoritmos recursivos**.

É importante entender o conceito de apontador: **uma variável criada no HEAP (dinâmica) só pode ser acessada por meio de um apontador**. Em tempo de **compilação** já se conhecem os endereços das variáveis estáticas declaradas. Vetores e matrizes, que podem ter vários elementos, têm uma **fórmula de cálculo do endereço efetivo** razoavelmente simples, baseando-se num endereço conhecido, o **endereço base** do vetor ou matriz. Mas com alocação dinâmica, não se pode afirmar nem quando nem quantos elementos serão alocados (pois isso depende da quantidade de dados que serão processados, o que não é conhecido em tempo de compilação) e, portanto, **não se pode calcular o endereço antes da execução e da alocação** propriamente dita do elemento. Assim, é preciso que, quando o elemento é alocado, o seu endereço seja devolvido (passe a ser conhecido), de maneira que possa ser usado pelo programa.

Quando você cria um objeto em Java ou C#, usando **new**, faz a alocação dinâmica de uma região da memória para armazenar o objeto. A variável que recebe o resultado de **new** é o apontador para o objeto.

2. Ponteiros

“a Força”

1. Endereços de Memória e Ponteiros

A memória de um computador é organizada de maneira que cada posição tem um endereço único. Isso permite que, se o endereço for conhecido, seja possível armazenar e recuperar valores dessas posições.

Várias linguagens de programação fazem uso intensivo do conceito de **apontador de memória**, ou **ponteiro**. Em C, são usados para passagem de parâmetros por referência, assim como em Pascal (embora de forma “disfarçada”).

Em Java, ponteiros são usados sempre que se acessa um objeto ou quando se passa um objeto como parâmetro para um método.

Ponteiros podem ser usados para várias atividades, como acesso a vetores e matrizes sem o uso de indexação, o que torna o acesso mais rápido, porque evita o cálculo de endereço efetivo da posição que se deseja acessar; esse cálculo usa multiplicação, que é uma operação relativamente lenta. Ponteiros também são usados para a passagem de parâmetros entre métodos de um objeto.

Toda variável possui, portanto, um endereço que é apenas seu. Em geral, um endereço se refere a uma única variável. Portanto, quando um programa armazena um valor em uma variável, ele nada mais faz do que copiar bits de um endereço de origem (local da memória) para o endereço de destino (outro local da memória), que é o endereço da variável receptora.

Um **ponteiro** é uma variável que **contém o endereço** de memória de **outra variável**. Se o conteúdo da variável A é o endereço da variável B, dizemos que A **aponta** para B, ou A **referencia** B. Isso é ilustrado na figura abaixo:

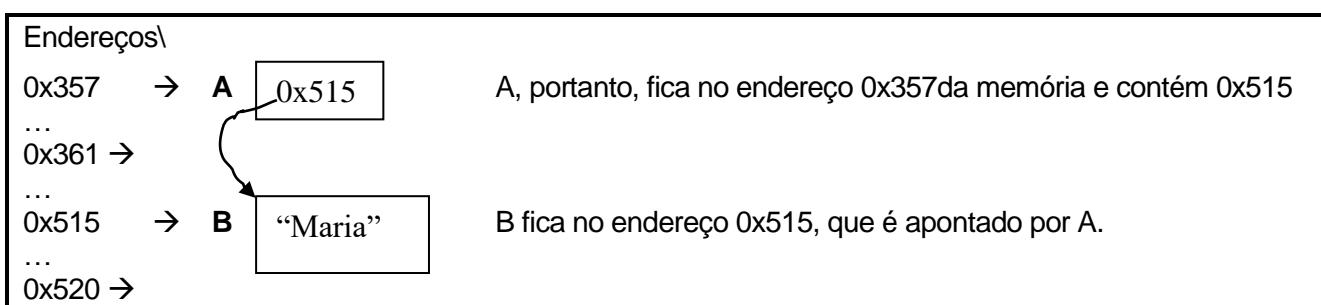


Figura 2 – endereços de memória e ponteiros

Ponteiros em C# e Java

Em Java e C#, ponteiros são usados implicitamente sempre que um objeto é instanciado a partir de uma classe e, também, quando um objeto instanciado é acessado para obter o valor de um de seus atributos ou invocar um de seus métodos, como vemos abaixo:

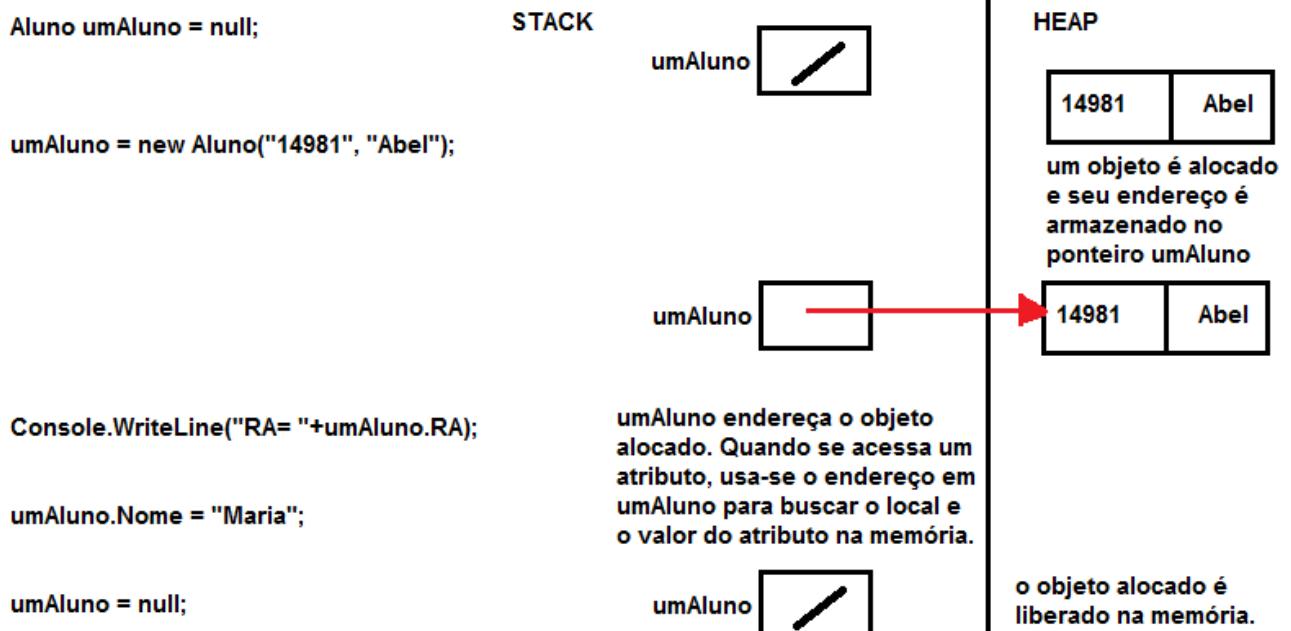


Figura 3 - mecanismo de ponteiros para Objetos em C# e Java

3. Listas Ligadas

1. Conceitos fundamentais

Uma **lista linear de elementos** é um conjunto de dados que guardam entre si uma **relação de ordem**. Como exemplos, podemos citar listas telefônicas, um catálogo de livros, uma lista de chamada de uma classe, e muitos outros. Pode-se implementá-las num computador, por meio do armazenamento em vetores. As operações usuais que se faria com os dados seriam as de pesquisa (busca) de elemento, inclusão de elementos, exclusão e alteração de dados.

Supondo-se que os dados devem permanecer **ordenados** por alguma chave que faça parte de cada elemento da lista, nas operações de inclusão e exclusão existirá a necessidade de **deslocamento físico dos elementos**, para a criação ou liberação de espaço, no caso de uso de um vetor. Isto toma certo tempo de processamento, que depende diretamente do tamanho da lista e do número de bytes ocupados por cada elemento, além da posição do vetor em que será feita a reserva do espaço. Além disso, quando se deseja incluir novos elementos na lista, poderá ocorrer o problema de se necessitar de mais memória do que a disponível no vetor em que se armazenou a lista, pois vetores são alocados estaticamente, não podendo ter seu tamanho alterado após a declaração.

Listas ligadas são usadas para resolver esses problemas. Numa lista ligada, os elementos são dispostos de uma forma em que cada um tem uma ligação com o elemento seguinte da lista: em cada elemento existe um **campo que indica o próximo elemento**. Assim, os elementos individuais não são armazenados sequencialmente de forma física, mas de forma lógica, pois é a seqüência de ligações que determina a ordem em que serão acessados, e não a ordem dos endereços que ocupam na memória, como ocorre em vetores e matrizes. Uma lista ligada tem, portanto, o aspecto dado na figura abaixo:

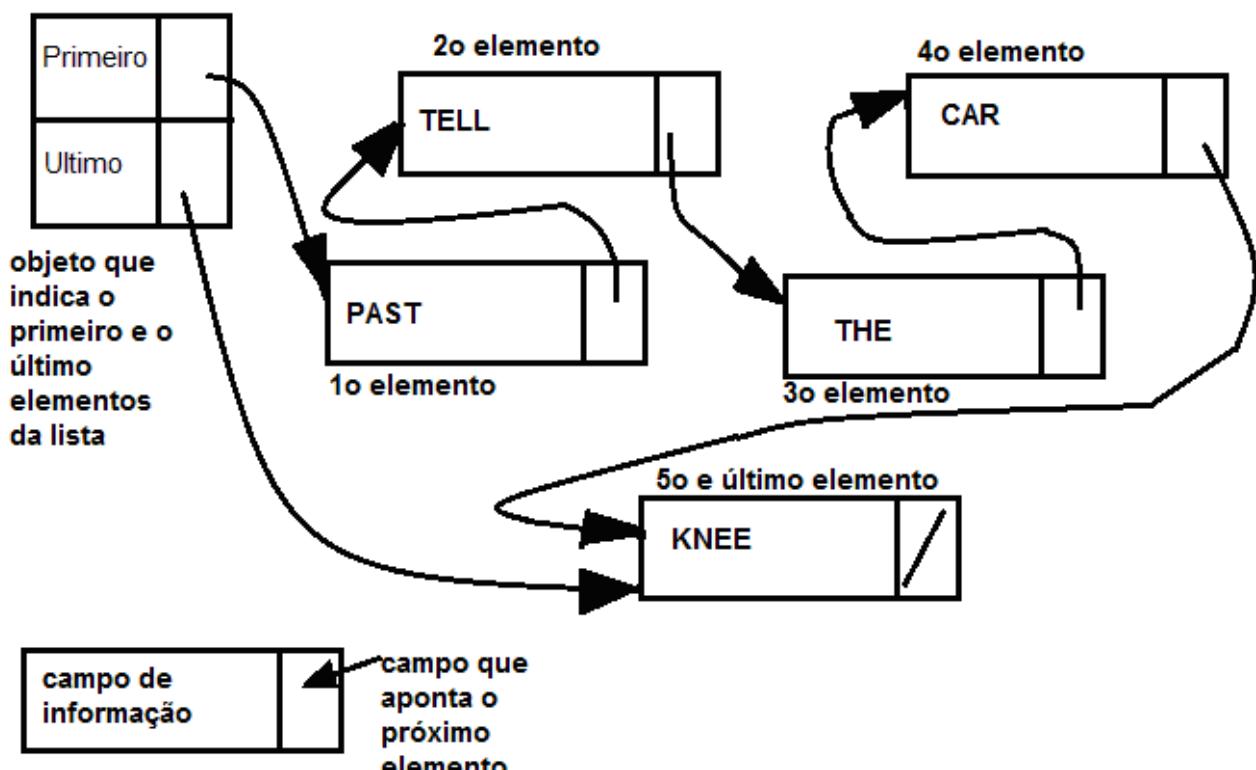


Figura 4 – uma lista ligada com 5 elementos

Observe, na figura 16, que o último elemento (o quinto) tem, em seu campo indicador do próximo elemento, um símbolo “ / “, que indica que a lista **termina nesse elemento**. Em outras palavras, o símbolo “ / “, neste contexto, informará que o apontador não indica nenhum endereço, ou seja, que vale **null** porque não há **nenhum** elemento **após** ele.

Cada elemento de uma lista ligada é um registro composto de 2 campos principais:

- O campo de **informação**, que armazena um objeto genérico com quaisquer informações
- O campo de **apontador para o próximo elemento** da lista.

O primeiro campo armazenará os dados a serem processados pelo programa. O outro campo armazenará o **endereço** do elemento **seguinte** da lista de dados.

Há também a necessidade de um objeto **que aponte o início e o final da lista ligada**, pois todas as operações de **pesquisa** serão feitas tomando-se o primeiro elemento como **ponto de partida**.

Para criar-se uma lista ligada, os dados são lidos e colocados na lista mas, para isso, é necessário **alocar**-se espaço para armazenar cada dado lido. O espaço alocado para cada elemento será doravante chamado de **Nó**. Geralmente, cada nó criado deve ser ligado ao último nó colocado anteriormente. Para isso, o campo **apontador do último nó deverá conter o endereço do novo nó**, de modo que o último nó aponte para este novo nó que, dessa maneira, fica ligado ao anterior.

O campo apontador de cada nó conterá o endereço do nó seguinte. O acesso é feito de forma **indireta**, ou seja, o **endereço de um nó** é dado pelo **conteúdo de uma variável apontadora**.

Por exemplo, se um apontador P tem como conteúdo o valor 54, isto significa que ele aponta para o **endereço 54 da memória**. O conteúdo da posição 54 é o **conteúdo do conteúdo do apontador P**.

Antes de passarmos ao manuseio de listas ligadas, convém adotarmos uma forma de representar os apontadores de cada elemento. Assim sendo, suponhamos que os dados lidos serão armazenados num registro com dois campos: **informacao** e **prox**, e que P é um apontador da lista de dados, como na figura abaixo:

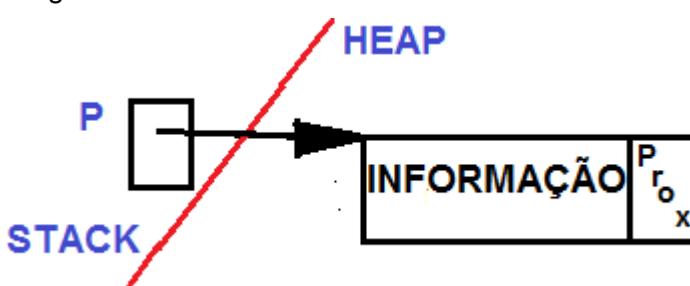


Figura 5 – componentes de um nó de lista ligada

Quando escrevermos **P**, isto indicará o **Conteúdo de P**, que é o **endereço do nó que P aponta**. Em C# e em Java, isso indica uma referência ao objeto apontado por P.

Escrevendo **P.informacao**, indicamos o **campo Informacao do nó apontado por P**.

Quando escrevermos **P.prox** indicaremos o campo **prox** do nó apontado por P, que nada mais é do que o **endereço do nó seguinte ao nó apontado por P**.

Podemos aplicar esses conceitos observando a figura abaixo:

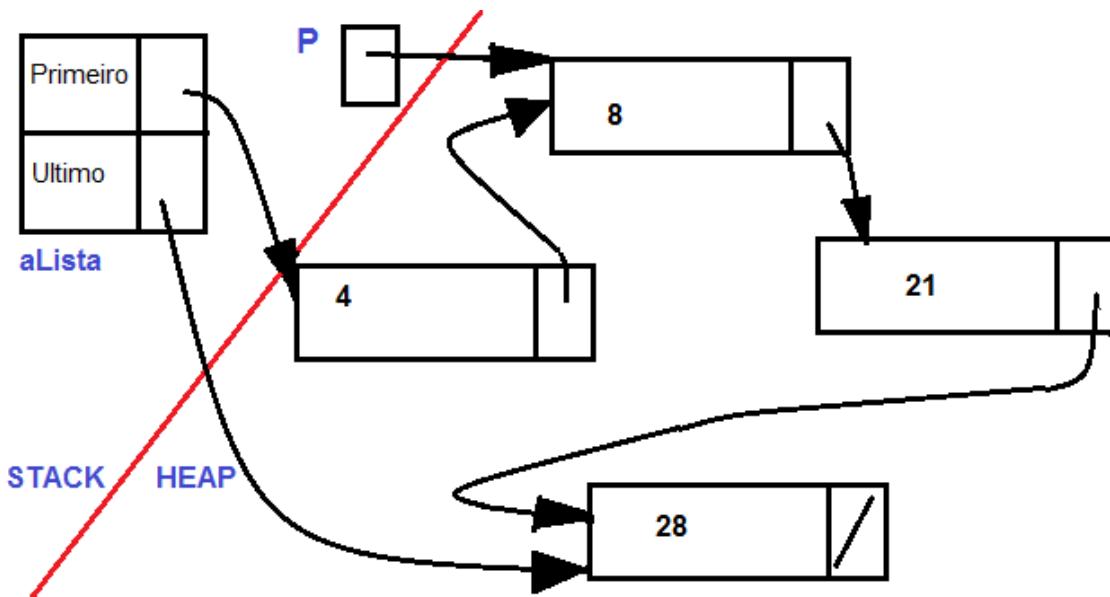


Figura 6 – acessando uma lista ligada

Na figura acima, a variável **P** aponta um elemento da lista ligada cujo campo de informação contém o valor 8. O objeto **aLista** aponta para o primeiro e para o último nós da lista. O próximo elemento a partir do nó apontado por P é o que contém o valor 21. Então,

$$P.info = 8$$

$$P.prox.info = 21$$

$$P.prox.prox.info = 28$$

P.prox.prox.prox.info acarreta erro, pois o elemento com valor 28 é o último da lista ligada; o campo PROX não aponta para posição alguma da memória

Note que não interessa qual o endereço específico de memória de cada elemento. Quando por exemplo, executamos o comando

$$P = P.prox;$$

o apontador **P** receberá o conteúdo do campo **PROX** do nó que ele aponta no momento. Em outras palavras, P passará a apontar para o elemento seguinte, de acordo com a figura a seguir.

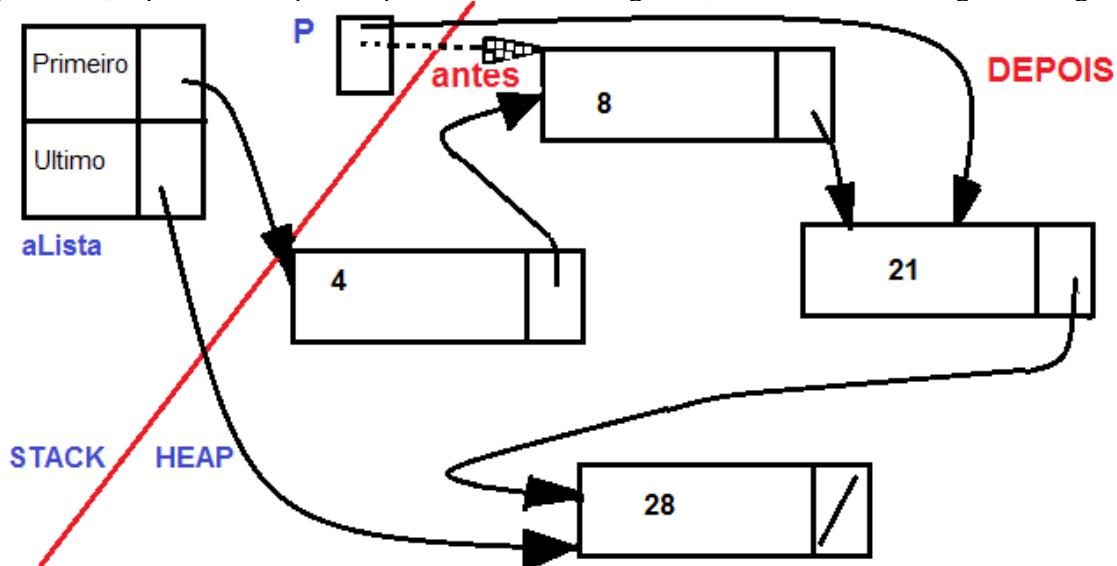


Figura 7 – percorrendo uma lista ligada

P passará a apontar o nó cujo campo Info vale 21. Assim, o valor de P.info valerá 21. Portanto, um comando da forma P = P.prox usado repetidas vezes faz com que os nós da lista ligada sejam "visitados" pelo apontador P, ou, em outras palavras, permite o acesso a cada nó, na seqüência em que estão ligados.

Para indicar o fim de uma lista ligada, usamos a constante pré-definida **null**, que indicaremos pelo símbolo **/**. O comando abaixo

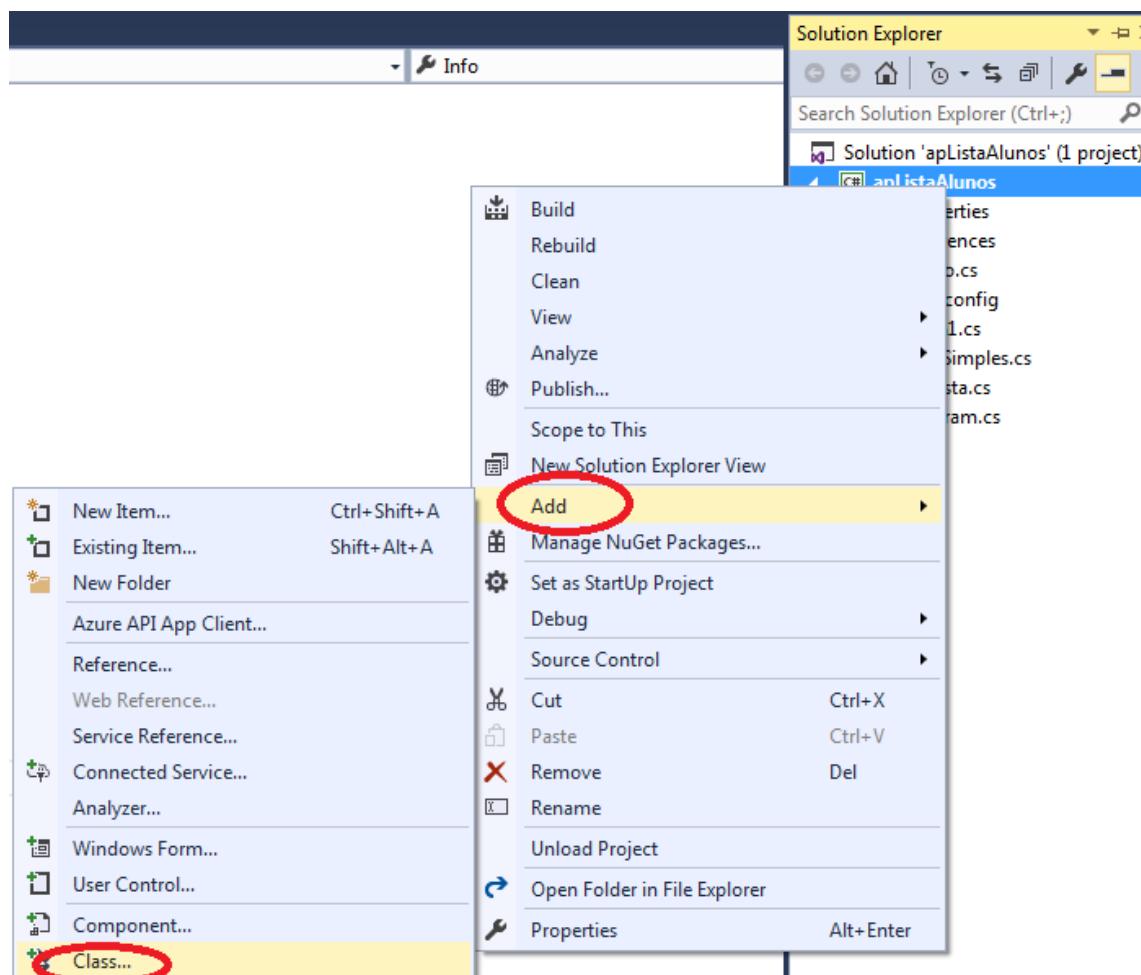
```
P = null;
```

faz com que o apontador P não aponte para lugar algum da memória, de modo que a referência P.info será o mesmo que **null.info**, o que acarreta uma exceção de acesso à memória, pois **null** não aponta para nada.

2. Declaração de classes para listas ligadas

Abaixo descrevemos uma classe NoLista, que define as características de um nó da lista ligada, e uma classe ListaSimples, que define o apontador para o início da lista e o número de nós presentes na lista a cada momento, bem como alguns métodos iniciais que estaremos discutindo a seguir.

No Visual Studio crie uma aplicação Windows Form, chamada apListaAluno. No Solution Explorer, clique com o botão direito no nome da solução, selecione [Add] e [Class...].



Crie uma nova classe e a chame de NoLista. Digite o código explicado a seguir.

```
public class NoLista<Dado> where Dado : IComparable<Dado>
{
    private Dado info;
    private NoLista<Dado> prox;
}
```

NoLista é uma classe genérica, que utiliza um parâmetro de tipo <Dado> para tornar possível a essa classe adiar a especificação de um ou mais tipos de dados até que a classe ou método seja declarada e instanciada pelo código da aplicação que usará essa classe. Dessa forma, a classe NoLista não fica "presa" a um tipo de dados específico (por exemplo, versões de NoLista feitas exclusivamente para instâncias da classe Aluno ou da classe Livro). Codificamos apenas uma versão de NoLista, genérica e, quando uma aplicação específica usar essa classe, essa aplicação definirá qual o tipo de dados que será usado no lugar do parâmetro de tipo <Dado>.

Observe também o uso da **Constraint where**, que especifica que Dado deverá ser uma classe que implementa a interface IComparable. Essa interface exige que a classe que a implementa possua o método CompareTo, que compara dois itens. Nesse caso, ainda, IComparable<Dado> indica que os itens a serem comparados serão, também, da classe que substituirá o parâmetro de tipo <Dado>.

Em outras palavras, se uma aplicação for utilizar a classe NoLista para processar informações de objetos da classe Aluno, a classe Aluno deverá implementar IComparable. A declaração de um NoLista, nessa **aplicação**, seria semelhante a:

```
NoLista<Aluno> umAluno = new NoLista<Aluno>(new Aluno("14981", "Abel"), null);
```

Vamos declarar uma **Propriedade** para o atributo info. Propriedades em C# e em Delphi substituem os métodos acessores Get e Set usados em Java:

```
public Dado Info
{
    get => info;
    set
    {
        if (value != null)
            info = value;
    }
}
```

Vamos também declarar uma propriedade para acessar o atributo prox, como vemos abaixo:

```
public NoLista<Dado> Prox
{
    get => prox;
    set => prox = value;
}
```

Você pode fazer com que o Visual Studio crie as propriedades para você, clicando com o botão direito no nome do atributo e selecionando a opção Quick Action, ilustrada abaixo. Em seguida, escolha a operação que deseja ser feita:

```

private Dado info;
private NoLista<Dado> prox;

Encapsulate field: 'prox' (and use property) ...
Encapsulate field: 'prox' (but still use field)
{
    get { return info; }
    set
    {
        if (value != null)
            info = value;
    }
}

public NoLista(Dado novaInfo, NoLista<Dado> proximo)
{
    Info = novaInfo;
    Prox = proximo;
}

internal NoLista<Dado> Prox
{
    get
    {
        return prox;
    }
    set
    {
        prox = value;
    }
}

```

Agora codificamos o construtor da classe:

```

public NoLista(Dado novaInfo, NoLista<Dado> proximo)
{
    Info = novaInfo;
    Prox = proximo;
}

```

Em seguida, criaremos e codificaremos a classe `ListaSimples`, que também será genérica, especificando o mesmo parâmetro de tipo que a classe `NoLista`. Dessa maneira, a lista ligada encapsulará objetos `NoLista` que manipularão objetos da classe `Dado`.

```

class ListaSimples<Dado> where Dado : IComparable<Dado>
{
    private NoLista<Dado> primeiro, ultimo, anterior, atual;
    int quantosNos;
    private bool primeiroAcessoDoPercorso;

    public ListaSimples()
    {
        primeiro = ultimo = anterior = atual = null;
        quantosNos = 0;
        primeiroAcessoDoPercorso = false;
    }

    public void PercorrerLista()
    {
        atual = primeiro;
        while (atual != null)
        {
            Console.WriteLine(atual.Info);
            atual = atual.prox;
        }
    }

    public bool EstaVazia
    {

```

```

        get => primeiro == null;
    }

    public NoLista<Dados> Primeiro
    {
        get => primeiro;
    }

    public NoLista<Dados> Ultimo
    {
        get => ultimo;
    }

    public int QuantosNos
    {
        get => quantosNos;
    }
}

```

Vamos criar também o método `InserirAntesDoInicio()`. Sua função é incluir um novo nó antes do primeiro nó da lista. Ele receberá como parâmetro um apontador para um nó da lista (uma instância de `NoLista`). Portanto, antes de chamar esse método, a aplicação que usa a lista ligada deverá já ter criado um nó de lista (classe `NoLista`) e também o objeto que será armazenado no atributo `info` desse nó, para podermos incluí-lo na lista.

No entanto, devemos pensar em alguns casos específicos que podem ocorrer durante a inclusão de um novo nó antes do primeiro da lista. Se fosse um vetor, teríamos de deslocar todos os elementos, desde o último até o primeiro, para a posição seguinte do vetor.

Com uma lista ligada, basta fazermos o novo nó apontar para o que atualmente é o primeiro. Mas, e no caso de uma **lista vazia**? Não haveria um nó anterior na lista. Nesse caso, o atributo `primeiro` do objeto `ListaSimples` valerá `null`, de modo que o campo `prox` do novo nó receberá `null` e passará a ser o primeiro da lista. Também nesse caso o apontador para o último nó da lista terá seu valor atribuído, como vemos na descrição passo a passo abaixo:

1. O método recebe um objeto contendo o dado que será incluído como o conteúdo do nó inicial da lista ligada:

```
public void InserirAntesDoInicio(Dado novoDado)
{
```

3. Devemos instanciar um nó da lista ligada, para armazenar o dado passado como parâmetro. O construtor abaixo de `NoLista` armazena no atributo `info` o dado passado como parâmetro:

```
var novoNo = new NoLista<Dados>(novoDado);
```

Lista	
primeiro	null
ultimo	null
quantosNos	0



Figura 8 – uma lista vazia e um nó a ser incluído

4. Antes de incluir o nó `novoNo` na lista, precisamos verificar se a lista está vazia. Caso esteja, a primeira operação a ser feita é atribuir o valor de `novoNo` para o apontador do último nó da lista:

```
if (EstaVazia)                // se a lista está vazia, estamos
    ultimo = novoNo;           // incluindo o 1º e o último nós!
```

Como a lista está vazia nesse momento, a propriedade **EstaVazia** devolve **true** e, em seguida, o ponteiro do último nó da lista receberá o endereço do novoNo, como abaixo:

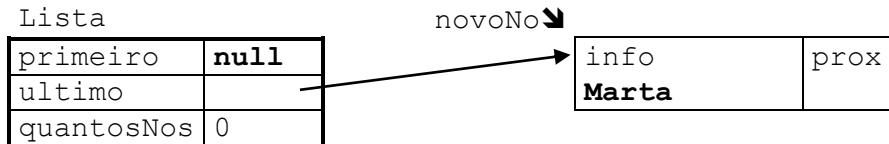


Figura 9 – o apontador do último nó aponta para o novo nó

5. Em seguida, é executado o comando `novoNo.prox = primeiro;`. Nesse momento, são feitas duas operações: obtém-se o valor de `primeiro` (que vale `null` neste momento) e atribui-se esse valor ao campo `prox` de `novoNo`, de modo que a configuração fica sendo a seguinte:

```
novoNo.Prox = primeiro; // faz o novo nó apontar para o nó
```

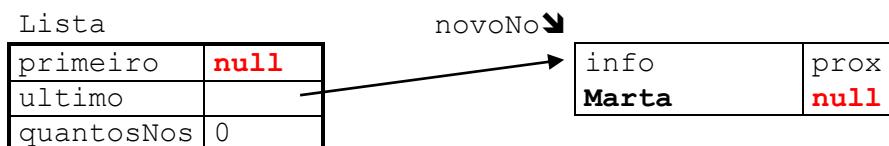


Figura 10 – o novo nó aponta para o primeiro nó, que vale **null**

6. Após isso, o atributo `primeiro` da lista ligada recebe o valor do apontador `novoNo`, de maneira que também passa a apontar o novo objeto `NoLista`, e o atributo `quantosNos` é incrementado:

```
primeiro = novoNo;           // atualmente no início da lista
quantosNos++;               // (que pode ser null)
}
```



Figura 11 – a inclusão termina, com o primeiro e último nós apontando para o nó recentemente incluído

Agora vamos inserir um novo nó nessa lista. Novamente, a aplicação que invoca o método `InserirAntesDoInicio()` deverá passar como parâmetro o Dado que se deseja colocar na lista (no atributo `info` do primeiro nó). Teríamos a seguinte configuração da memória logo após a chamada a esse método e a instanciação do novo nó:

```
var novoNo = new NoLista<Dados>(novoDado);
```

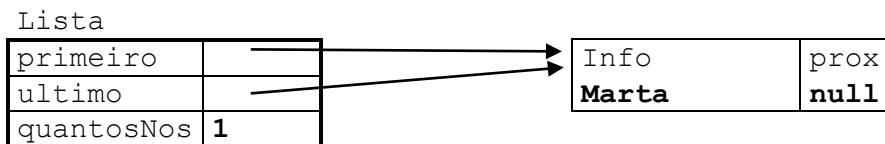


Figura 12 – incluindo mais um nó na lista ligada

O primeiro elemento é aquele apontado pelo atributo **primeiro** da Lista, cujo valor de **info** é “Marta”. Sigamos o método passo a passo, novamente, e notemos que, como nesse momento a lista não

está vazia, o comando **if** abaixo resultará em **false** e o comando **ultimo = novoNo** não será executado. Dessa forma, o último nó continua sendo esse que já faz parte da lista e que tinha sido incluído anteriormente.

```
if (estaVazia)           // se a lista está vazia, estamos
    ultimo = novoNo;      // incluindo o 1º e o último nós!
```

Como o **if** acima resultou em **false**, o fluxo de execução parte para o comando seguinte, e faz com que o campo **prox** do nó apontado por **novoNo** receba o mesmo endereço para o qual aponta o apontador **primeiro**. Dessa maneira, o novo nó aponta para o que era considerado o primeiro nó da lista que, por sua vez, agora passará a ser o segundo nó, como vemos a seguir:

```
novoNo.Prox = primeiro; // faz o novo nó apontar para o nó
```

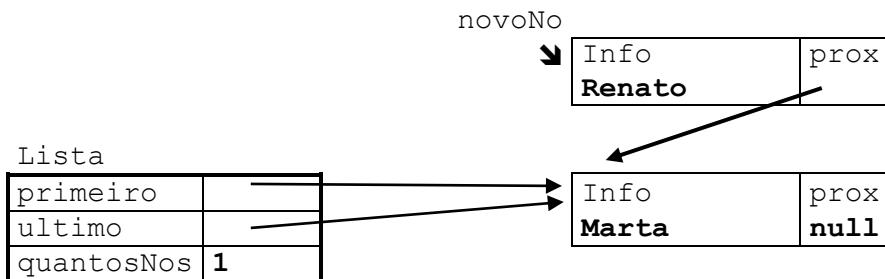


Figura 13 – o novo nó aponta para o primeiro nó, que neste momento aponta um nó já presente na lista

Em seguida, o atributo **primeiro** da lista ligada recebe o valor do apontador **novoNo**, de maneira que também passa a apontar esse objeto NoLista, e o atributo **quantosNos** é incrementado:

```
primeiro = novoNo;          // atualmente no início da lista
quantosNos++;               // (que pode ser null)
}
```

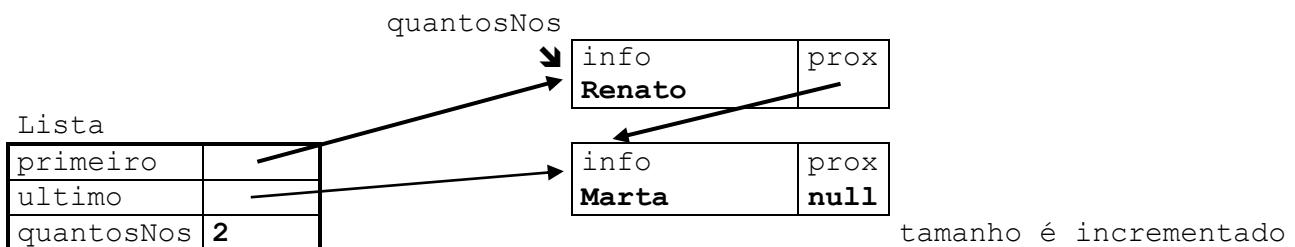


Figura 14 – a lista fica completa, com o primeiro nó apontando para o nó recentemente incluído

Se incluirmos mais um elemento nessa lista, teremos os seguintes passos, ilustrados abaixo:

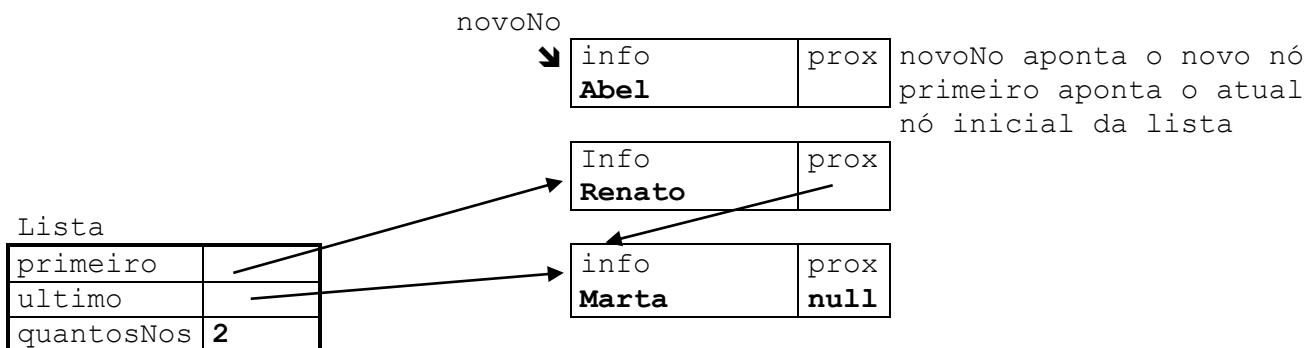


Figura 15 – (a) iniciando a inclusão do terceiro nó

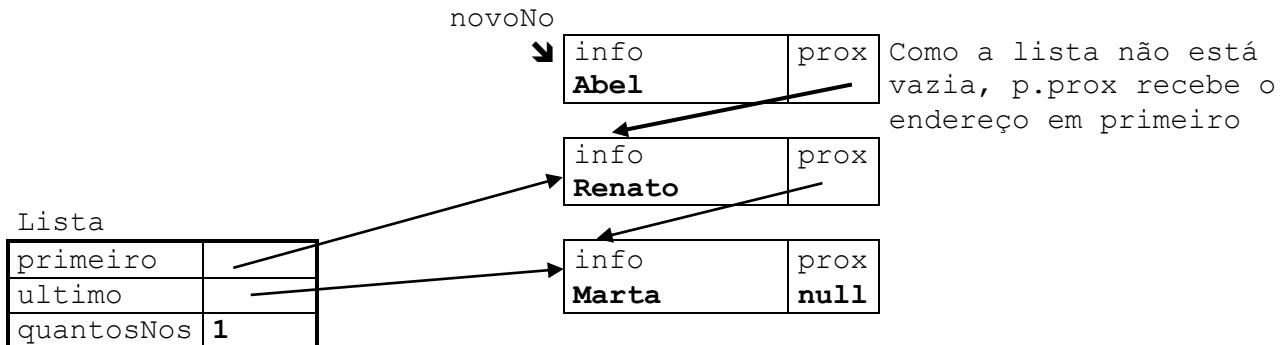


Figura 16 – (b) ligando o novo nó ao que atualmente é o primeiro nó

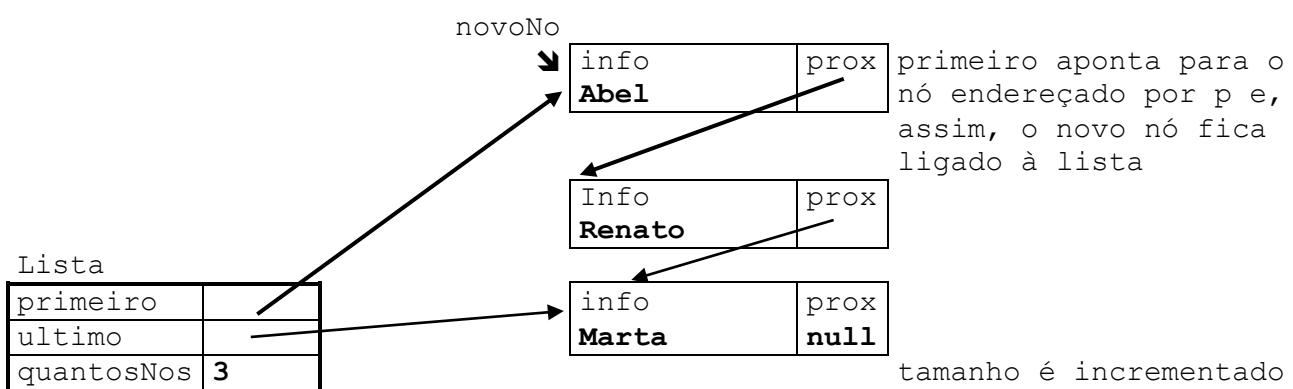


Figura 17 – (c) o primeiro nó aponta para o novo nó, após este ter sido ligado à lista

Para declarar variáveis apontadores em uma **aplicação**, na declaração dessas variáveis indicaremos uma classe específica que substituirá a classe genérica Dado, como abaixo:

```
public NoLista<Aluno> pri, p, atual, anterior;
```

Na declaração acima as variáveis pri, p, atual, anterior são do mesmo tipo, ou seja, são apontadores de objetos do tipo NoLista<Aluno>. <Aluno> substitui, na aplicação, o parâmetro de tipo <Dado> e, com isso, supomos que nessa aplicação se trabalhará com uma lista ligada onde cada nó armazena, no atributo **info**, uma instância da classe Aluno.

A classe Aluno deverá implementar a interface IComparable, para atender as restrições ([constraints](#)) da declaração da classe NoLista<Dado>, onde se informa que a classe específica que substituir a classe genérica Dado deverá implementar IComparable:

```
public class NoLista<Dado> where Dado : IComparable<Dado>
```

Já que a alocação de um NoLista<Dado> é feita em tempo de execução, a posição de memória que será o endereço do nó alocado só será conhecida no momento da alocação. Esse endereço poderá ser qualquer um disponível na memória e, por isso, deverá ser armazenado numa variável, para que se possa acessar o nó, indiretamente, quando for necessário. Esta variável deverá ser um apontador do tipo do objeto referente ao nó alocado.

Em C# e Java, o comando que faz a alocação dinâmica é o **new**, que chama o método construtor do objeto que se deseja instanciar (criar na memória) e devolve o endereço de memória do nó (objeto) criado. A forma para executá-lo é a seguinte:

```
p = new NoLista<Aluno>(new Aluno("14981", "Abel"), null);
```

Este comando fará com que um objeto da classe NoLista seja alocado, e seu endereço seja armazenado em p. Na alocação do objeto NoLista, o construtor da classe será chamado.

3. Aplicação usando Lista Ligada

No formulário da nossa aplicação de exemplo, vamos criar a interface visual para o usuário executar tarefas com a lista ligada. O ideal é que nosso usuário no tenha conhecimento de que está usando uma estrutura de dados específica mas, pelo contrário, que a codificação e a operação do programa sejam o mais isoladas possível dos detalhes internos das estruturas de dados escolhidas.

Vamos primeiramente criar uma classe aluno como a que se segue:

```
namespace apListaAlunos
{
    class Aluno : IComparable<Aluno>
    {
        const int tamanhoRA = 5;
        const int tamanhoNome = 30;
        const int tamanhoNota = 4;
        const int inicioRA = 0;
        const int inicioNome = inicioRA+tamanhoRA;
        const int inicioNota = inicioNome+tamanhoNome;

        string ra, nome;
        float nota;

        public string RA      // propriedade
        {
            get => ra;
            set {
                if (value != "")
                    ra = value.Substring(0, tamanhoRA).PadLeft(tamanhoRA, '0');
                else
                    throw new Exception("RA vazio é inválido");
            }
        }

        public Aluno(string linhaDeDados)
        {
            Ra    = linhaDeDados.Substring(inicioRA, tamanhoRA);
            Nome = linhaDeDados.Substring(inicioNome, tamanhoNome);
            Nota = double.Parse(linhaDeDados.Substring(inicioNota, tamanhoNota));
        }

        public Aluno(string ra, string nome, float nota)
        {
            Ra    = ra;
            Nome = nome;
            Nota = nota;
        }

        public Aluno(string ra)
        {
            Ra = ra;
        }

        public int CompareTo(Aluno outroAluno )
        {
            return ra.CompareTo (outroAluno.ra);
        }

        public override string ToString() {
            return ra+" "+nome+" "+nota;
        }
    }
}
```

Observe que essa classe implementa a interface IComparable, através da codificação do método CompareTo, que recebe como parâmetro um objeto da classe Aluno. Assim, a classe Aluno atende às constraints estabelecidas nas classes ListaSimples e NoLista, de forma que as classes em estudo podem armazenar objetos da classe Aluno.

Temos 3 construtores, que poderão ser usados em diversos momentos, de acordo com a origem dos dados que serão usados para instanciar um objeto da classe Aluno. O primeiro construtor recebe como parâmetro uma string apenas, que conteria uma linha de dados lida de um arquivo texto. Essa linha será separada nos campos constituintes de um registro de alunos nesse arquivo e cada campo será armazenado no atributo correspondente, através do acessador set de cada propriedade referente a um atributo.

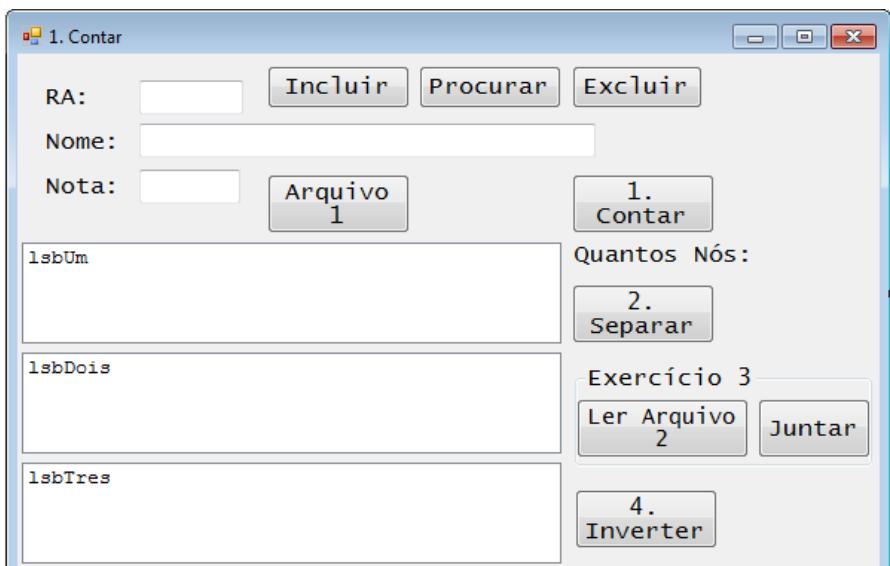
O segundo construtor recebe como parâmetro os valores individuais de cada atributo; esses valores serão obtidos separadamente pela aplicação e, quando um objeto de classe Aluno tiver de ser instanciado para armazenar esses valores, será chamado o segundo construtor com os valores passados como parâmetros.

Já o terceiro construtor recebe apenas o RA do aluno como parâmetro. Ele será útil em situações em que apenas o RA do aluno é necessário, como efetuar pesquisas na lista para exclusão ou exibição.

Vamos realizar o design de nosso formulário, de acordo com a figura ao lado:

Esse design nos permitirá também realizar alguns dos exercícios deste capítulo. Os três textBox são chamados, respectivamente, de txtRA, txtNome e txtNota. Possuem a propriedade MaxLength igual a 5, 30 e 5, respectivamente. Use nomes adequados para os demais componentes visuais.

No início do código do formulário, declare duas listas ligadas, como abaixo:



```
public partial class Form1 : Form
{
    Listasimples<Aluno> lista1, lista2;
```

No evento Form_Load do formulário, faremos a instanciação de nossa primeira lista ligada de alunos, com o comando abaixo:

```
private void Form1_Load(object sender, EventArgs e)
{
    lista1 = new Listasimples<Aluno>();
```

No evento Click do botão "Arquivo 1", digite o código abaixo, que lerá um arquivo texto com os dados dos alunos e os armazenará na lista ligada:

```
private void btnLerArquivo1_Click(object sender, EventArgs e)
{
    fazerLeitura(ref lista1, lsbUm);
```

O método fazerLeitura recebe como parâmetro por referência a lista ligada que será montada a partir da leitura dos dados de um arquivo. O segundo parâmetro é um ponteiro para o listBox que será usado para exibir os dados lidos. No caso acima, faremos a leitura e montagem da lista1 e a exibição de seus dados no listBox lsbUm. O código abaixo usa um OpenFileDialog para que o usuário informe o local e nome do arquivo que será lido. Portanto, vá até a janela ToolBox, na coleção Dialogs, e arraste um OpenFileDialog para seu formulário, e mude seu nome para dlgAbrir na janela Properties.

```
private void fazerLeitura(ref ListaSimples<Aluno> qualLista, ListBox qualListBox)
{
    qualLista = new ListaSimples<Aluno>(); // recria a lista a ser lida

    if (dlgAbrir.ShowDialog() == DialogResult.OK)
    {
        StreamReader arquivo = new StreamReader(dlgAbrir.FileName);
        string linha = "";
        while (!arquivo.EndOfStream)
        {
            linha = arquivo.ReadLine();
            qualLista.inserirAposFim(new Aluno(linha));
        }
        qualLista.listar(qualListBox);
        arquivo.Close();
    }
}
```

Observe que não criamos ainda os métodos inserirAposFim e listar da classe ListaSimples. Você deve analisar o que fazem, a partir de seus parâmetros, e codificá-los na classe ListaSimples para poder testar o programa e fazer os exercícios abaixo.

Conforme avançarmos no estudo de listas ligadas, voltaremos a este programa para incorporar novas funcionalidades à lista e à nossa interface de aplicação.

4. Exercícios

1. Faça uma função que, após percorrer e contar todos os nós da lista, devolva o número de nós de uma lista ligada cujo apontador inicial está encapsulado no próprio objeto ListaSimples.
2. Faça um método que separe uma lista ligada de números inteiros em duas: uma com os números pares e outra com os ímpares. Os nós usados nas duas novas listas devem ser os nós originais (Não se deve criar outros). Lembre-se que cada lista ligada deve ter um apontador inicial próprio.
3. Faça um método que receba como parâmetros os apontadores iniciais de 2 listas ligadas ordenadas diferentes, de números inteiros, e que devolva uma nova lista, que é construída pelo casamento das duas outras, sem repetição de elementos, e liberando os repetidos.
4. Desenvolva um método que inverta uma lista ligada do tipo ListaSimples. Não é permitido usar outra lista, de forma que você deve trabalhar com ponteiros que permitam, durante o percurso da lista, fazer a inversão das ligações.

5. Busca em listas ligadas

Vamos supor que temos uma lista ligada, já criada, cujos nós representam informações provenientes de um arquivo de dados de alunos, e que desejamos procurar uma determinada informação dentro dessa lista.

No arquivo, existiam como valores o RA (Registro Acadêmico) e o Nome de cada aluno. O RA será considerado uma chave primária de acesso.

Uma chave primária é um campo que não tem repetições para nenhum dos registros de um arquivo. Dessa maneira, pelo RA podemos identificar univocamente cada aluno, ou seja, não existe um RA que seja usado para dois alunos distintos.

A figura a seguir ilustra a disposição de um trecho da lista ligada:

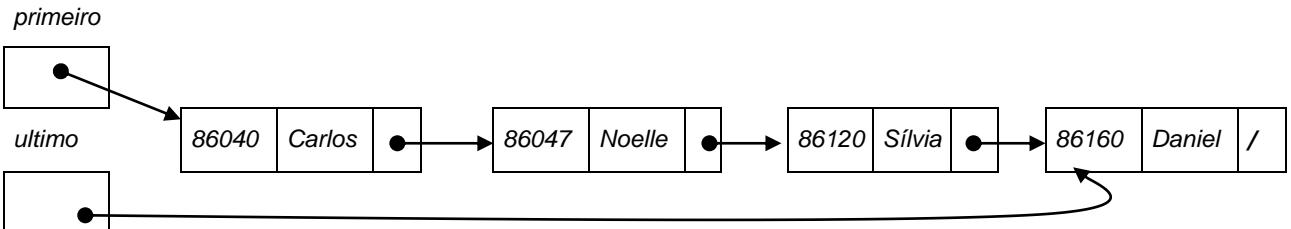


Figura 18 – uma lista ligada de alunos

Como você pode notar, cada nó possui um objeto de informação (elemento) formado por dois campos : o RA e o Nome do aluno. Observe que os dados da lista estão ordenados pelo campo RA.

Queremos desenvolver um algoritmo que procure um registro na lista, usando RA como chave de acesso. A ordenação facilitará a pesquisa.

Como se sabe, o acesso a cada nó deverá ser feito através de um apontador que indique o nó atual.

Primeiramente, convém se utilizar de outro apontador que não seja o apontador do **primeiro** nó da lista para acessar os nós durante a pesquisa do RA procurado, pois **primeiro** aponta para o início da lista, e se o "movimentarmos", perderemos a referência ao primeiro nó da lista, de forma que não mais poderemos usá-la após isto. Perdendo a referência ao primeiro nó de uma lista ligada, os nós seguintes serão perdidos e o mecanismo de gerenciamento de memória de Java acabará por liberar os nós, disponibilizando-os para outras aplicações.

Em segundo lugar, a pesquisa terá várias condições de parada. O RA procurado pode : (1) estar na lista, ou (2) não estar na lista. Em ambos os casos, a procura deve parar.

Mais precisamente, como a lista está ordenada crescentemente, se o RA procurado for menor que o RA do primeiro nó, podemos afirmar que seguramente o RA procurado não está na lista. De forma simétrica, se o RA procurado for maior do que o último RA da lista, o RA procurado não estará na lista.

Portanto, se isto não ocorrer, deve-se percorrer a lista, procurando, em cada nó, o RA desejado. Neste caso, temos:

- Se encontramos na lista um RA maior que o procurado, isto indica que o procurado não pertence à lista.
- Se o RA procurado for maior que todos os demais RAs presentes na lista ligada, a pesquisa irá até o final da lista, e o RA procurado não será encontrado. Se estes casos não ocorrerem, o RA estará na lista ligada, e portanto, deverá haver um apontador que o refencie. Assim, as condições de parada do algoritmo são:
 1. RA procurado < RA do **primeiro** nó
 2. RA procurado < RA do nó **atual**, onde **atual** aponta um nó qualquer da lista, após percorrê-la desde seu início
 3. **atual = null** (**atual** chegou ao fim da lista, sem encontrar o RA desejado)
 4. RA procurado = RA do nó **atual** (achamos o RA desejado)

Enquanto uma destas condições não ocorrer, a pesquisa deverá ser feita. Como você já deve ter percebido, numa lista ligada a pesquisa precisa ser seqüencial, ou seja, um elemento de cada vez, na seqüência das ligações entre os nós. Isso ocorre porque, para se acessar um nó, é preciso ter-se

acessado o nó anterior, que guarda a informação de como se chega ao nó desejado, ou seja, aponta para o seguinte. Abaixo temos uma figura mostrando esse processo:

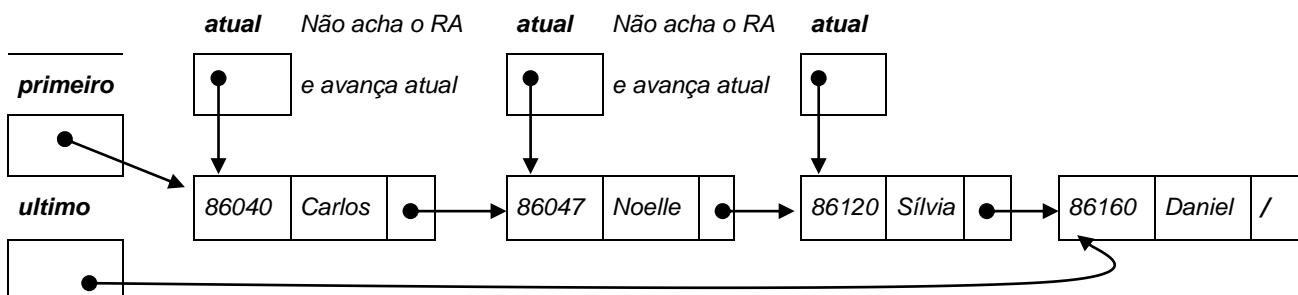


Figura 19 – o percurso sequencial na lista para procurar o ra 86120.

Quando se encontrar o elemento procurado, **atual** deverá conter seu endereço, para que se possa acessá-lo.

Mas, para podermos efetuar as comparações que citamos acima, observemos que a classe NoLista não possui o campo RA que, obviamente, teria de fazer parte da informação armazenada no campo **info** do NoLista. Da mesma maneira, o atributo **info** do NoLista não possui o campo RA diretamente mas sim encapsulado em um objeto da classe Aluno. Caso colocássemos o campo RA em NoLista, esta classe passaria a ser específica, ou seja, sempre teria esse campo e, portanto, não deveria ser usada para representar nós de outros tipos de lista ligada como, por exemplo, nós de Peças de Carro ou nós de Produtos.

Portanto, para manter a generalidade da classe NoLista e, ainda assim, permitir que ela possua um campo chave de pesquisa, informamos na declaração que o objeto armazenado em info deverá encapsular, obrigatoriamente, uma função que compara os dados armazenados num determinado nó. Essa função é o método **compareTo** e, para torná-lo obrigatório, faremos com que a classe NoLista, que armazena um objeto cuja classe é definida pelo parâmetro de tipo, apenas aceite classes que implementem a Interface **IComparable**. Essa interface é previamente definida em Delphi, C# e Java, e exige a implementação do método **compareTo**, que recebe como parâmetro o objeto que será comparado com o objeto atual (**this**).

Portanto, o objeto que for armazenado no atributo **info** de NoLista, seja de qual classe for, deverá implementar a interface **IComparable** ou seja, é obrigado a implementar o método **CompareTo**, o que já fizemos quando codificamos a classe **Aluno**, pois ela atende esse requisito de NoLista.

Os atributos **atual** e **anterior** serão usados na pesquisa sequencial de dados na lista. Vamos criar propriedades para permitir o acesso a esses dois atributos.

CompareTo(), por definição, devolve os seguintes valores:

valor menor que < 0	se this é menor que o outro objeto
valor igual a 0	se this é igual ao outro objeto
valor maior que 0	se this é maior que o outro objeto

Nossa aplicação de exemplo deverá, portanto, permitir buscar alunos através do campo RA, que usamos como chave de comparação no método **CompareTo** da classe **Aluno**.

Nossa aplicação, até o momento, define botões para disparar as operações que gostaríamos de fazer com a lista. Por exemplo, montar a lista a partir de um arquivo texto e chamar o método que efetua a pesquisa de uma chave na lista, para inserir novos alunos, mostrar dados de alunos já armazenados na lista e excluir os dados de um aluno cujo RA seja digitado e esteja armazenado na lista.

O método que faz a pesquisa na lista deve ser um método da classe **ListaSimples**. Abaixo temos o código do método **existeChave()**, para discussão. Esse código deve ser inserido no final de **ListaSimples**.

A primeira observação importante é sobre a configuração dos campos **atual** e **anterior** com **null**.

```

public bool ExisteDado(Dado outroProcurado)
{
    anterior = null;
    atual = primeiro;

    // Em seguida, é verificado se a lista está vazia. Caso esteja, é
    // retornado false ao local de chamada, indicando que a chave não foi
    // encontrada, e atual e anterior ficam valendo null

    if (estaVazia)
        return false;

    // a lista não está vazia, possui nós

    // dado procurado é menor que o primeiro dado da lista:
    // portanto, dado procurado não existe

    if (outroProcurado.CompareTo(primeiro.Info) < 0)
        return false;

    // dado procurado é maior que o último dado da lista:
    // portanto, dado procurado não existe

    if (outroProcurado.CompareTo(ultimo.Info) > 0)
    {
        anterior = ultimo;
        atual = null;
        return false;
    }

    // caso não tenha sido definido que a chave está fora dos limites de
    // chaves da lista, vamos procurar no seu interior

    // o apontador atual indica o primeiro nó da lista e consideraremos que
    // ainda não achou a chave procurada nem chegamos ao final da lista

    bool achou = false;
    bool fim = false;

    // repete os comandos abaixo enquanto não achou o RA nem chegou ao
    // final da lista

    while (!achou && !fim)

        // se o apontador atual vale null, indica final da lista

        if (atual == null)
            fim = true;

        // se não chegou ao final da lista, verifica o valor da chave atual
        else

            // verifica igualdade entre chave procurada e chave do nó atual

            if (outroProcurado.CompareTo(atual.Info) == 0)
                achou = true;
            else

                // se chave atual é maior que a procurada, significa que
                // a chave procurada não existe na lista ordenada e, assim,
                // termina a pesquisa indicando que não achou. Anterior
                // aponta o anterior ao atual, que foi acessado por
                // último

                if (atual.Info.CompareTo(outroProcurado) > 0)
                    fim = true;
                else
    }
}

```

```

// se não achou a chave procurada nem uma chave > que ela,
// então a pesquisa continua, de maneira que o apontador
// anterior deve apontar o nó atual e o apontador atual
// deve seguir para o nó seguinte

anterior = atual;
atual = atual.Prox;
}

// por fim, caso a pesquisa tenha terminado, o apontador atual
// aponta o nó onde está a chave procurada, caso ela tenha sido
// encontrada, ou o nó onde ela deveria estar para manter a
// ordenação da lista. O apontador anterior aponta o nó anterior
// ao atual

return achou; // devolve o valor da variável achou, que indica
               // se a chave procurada foi ou não encontrada
}

```

O método acima foi escrito sob a forma de uma função lógica, mas nada impede que seja escrito de outras formas. Note que ele só funciona corretamente com uma lista ordenada. Note também que no caso de uma lista ligada vazia, ele também funciona.

6. Inclusão de Novo Nó em lista ligada ordenada

Suponha que se deseja colocar um novo nó na lista ligada ordenada, de forma que esta ainda se mantenha ordenada. Há vários locais onde o novo nó pode ser colocado (ligado):

1. No início da lista, no caso de lista vazia
2. No início da lista, no caso do RA novo ser **menor que o primeiro** da lista;
3. Ao final da lista, no caso de o RA novo ser **maior que os demais**.
4. Entre dois **nós internos** da lista ligada.

Simplificadamente, deve-se procurar o intervalo no qual o novo nó se encaixaria, de forma ordenada, entre dois nós, e então ligar os três nós de forma correta, segundo o esquema abaixo:

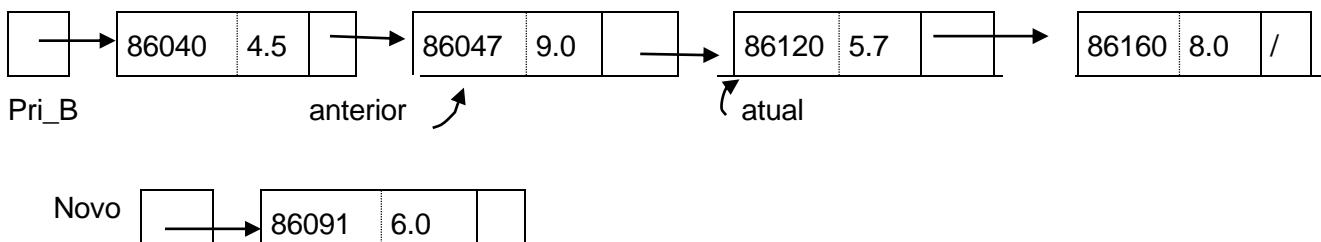


Figura 20 – Definindo o trecho onde incluir um novo nó em uma lista ordenada

Na figura acima, a pesquisa encontrou o local de inclusão do nó **NOVO**. Este deverá ser colocado entre o nó apontado por **ANTERIOR** e o nó apontado por **ATUAL**. Para tanto, o nó **ANTERIOR** deve apontar para **NOVO** e este para **ATUAL**, como indicado abaixo:

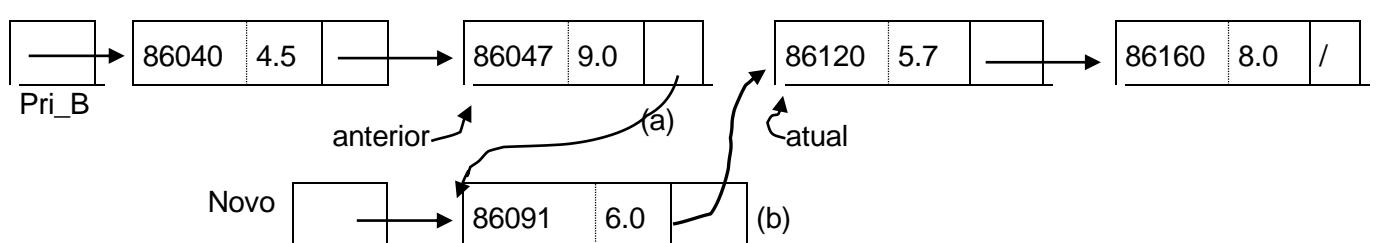


Figura 21 – Refazendo as ligações para inclusão de um novo nó na lista ligada

Deseja-se incluir o nó apontado por Novo (o novo nó). Procura-se o intervalo entre dois nós, no qual Novo^{RA} fique colocado ordenadamente. Agora, deve-se fazer com que o nó anterior aponte para o novo nó (em (a)) e que este aponte para o nó seguinte (em (b)), desfazendo a ligação anterior, de forma que a lista assuma o aspecto seguinte:

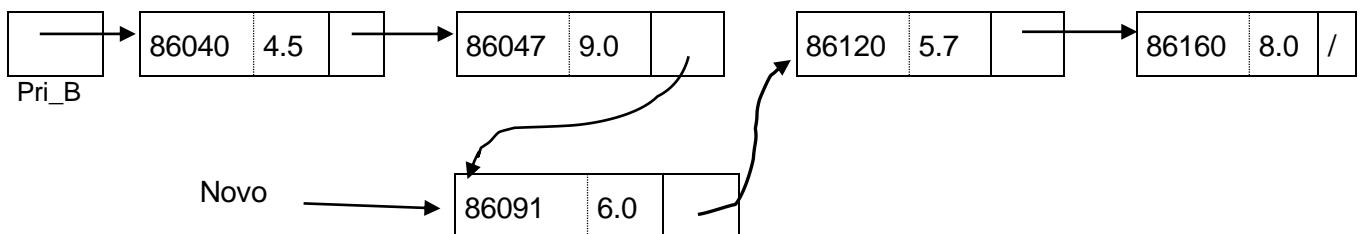


Figura 22 – Aspecto final da lista

Assim, ao percorrermos essa lista, a sequência dos RAs será: 86040, 86047, 86091, 86120, e 86160.

Como se nota, são necessários 4 apontadores para o processo acima: um para o início da lista, um para o novo nó, um para apontar o nó anterior e outro para apontar o nó posterior, em relação ao local de inclusão do novo nó.

A seguir, vem um algoritmo que inclui um novo nó numa lista ligada, em ordem da sua chave.

```
public void InserirEmOrdem(Dado dados)
{
    if (!existeDado(dados)) // existeChave configura anterior e atual
    {
        // aqui temos certeza de que a chave não existe
        // guarda dados no novo nó
        var novo = new NoLista<Dados>(dados, null);
        if (estaVazia) // se a lista está vazia, então o
            inserirAntesDoInicio(novo); // novo nó é o primeiro da lista
        else
            // testa se nova chave < primeira chave
            if (anterior == null && atual != null)
                inserirAntesDoInicio(novo); // liga novo antes do primeiro
            else
                // testa se nova chave > última chave
                if (anterior != null && atual == null)
                    inserirAposFim(novo);
                else
                    inserirNoMeio(novo); // insere entre os nós anterior e atual
    }
}

private void InserirNoMeio(NoLista<Dados> novo)
{
    // existeDado() encontrou intervalo de inclusão do novo nó

    anterior.Prox = novo; // liga anterior ao novo
    novo.Prox = atual; // e novo no atual

    if (anterior == ultimo) // se incluiu ao final da lista,
        ultimo = novo; // atualiza o apontador ultimo
    quantosNos++; // incrementa número de nós da lista
}
```

O apontador **anterior** está sempre apontando para um nó "atrás" do nó apontado por **atual**, de modo que, se o nó que desejamos incluir possuir uma chave maior que a maior chave atualmente presente na lista, **anterior** apontará o último nó e **atual** apontará **null**.

Se a lista for vazia, então o novo nó será o primeiro, e o método insereNoInicio se encarrega de também atualizar o apontador para o último nó.

7. Criação de uma lista ligada ordenada

Com os algorítmos vistos acima já é possível criar-se uma lista ligada ordenada, especialmente devido ao algorítmo de inclusão. Para criar uma lista ligada, basta ler os dados, sequencialmente, de um arquivo e chamar o método inserirEmOrdem(), passando por parâmetro os dados lidos.

O processo de ler e incluir nós ordenadamente pode demorar muito, pois ele obriga a um percurso na lista para a inclusão de cada novo nó. Seria muito mais rápido se houvesse uma leitura e montagem da lista original, sem se importar com ordem, e após a leitura fazer-se a ordenação.

Para montar uma lista ligada sem se preocupar com a ordem, deve-se ler cada registro, e ir-se colocando os nós correspondentes após o último elemento colocado na lista. Portanto, deve-se ter um apontador para o início da lista (sempre) e um para o último elemento nela colocado.

Assim, cria-se o nó, e se for o primeiro da lista, faz-se o apontador de início da lista apontar para ele. Então, o apontador de último nó recebe o mesmo endereço, ou seja, o valor do apontador inicial. Isto é feito, obviamente, porque o último nó também é o primeiro, quando este é colocado na lista.

Após isto, a cada novo nó que se cria, o campo de próximo do último nó aponta para este novo nó. O apontador último deve passar então a conter o endereço do novo nó. Ao final da criação, o campo **Próximo do último nó** deverá conter o valor **null**, pois é o fim da lista.

O procedimento que efetua esta operação acima descrita fica como **exercício**. Após isto, faça um procedimento que ordene a lista ligada. Note que você deve apenas trocar os apontadores, e não o conteúdo de informação de cada nó.

8. Ordenação de Listas Ligadas

Um método interessante para ordenar uma lista ligada é o seguinte:

1. Cria-se uma lista vazia, que ao final do processo será a lista ordenada
2. Percorre-se a lista original, não ordenada, e encontra-se o elemento com a menor chave da lista. Deve-se guardar o apontador para o nó anterior e para o nó onde se encontra o menor elemento
3. Remove-se o menor elemento da lista original (para isso usam-se os apontadores **anterior** e **atual**)
4. Inclui-se ao final da lista ordenada o nó recém-retirado da lista original
5. Repete-se as operações 2 a 5 enquanto houver nós na lista original

Utilizando esse método, você não tem necessidade de criar novos nós para uma lista ordenada, utilizando aqueles que já tinham sido alocados anteriormente para a lista desordenada.

9. Exclusão de um nó de uma lista ligada

Para excluir um nó de uma lista ligada, precisamos descobrir se esse nó existe, e indicar um apontador para ele. O método existeChave() estudado acima faz essa operação, e ainda indica o apontador para o nó anterior ao que desejamos excluir, o que é bastante importante, para ligar o nó anterior ao que queremos excluir com o nó posterior a este. Dessa maneira, a lista fica ligada corretamente.

No entanto, devemos sempre verificar se o nó que excluímos é o primeiro e/ou o último nós da lista, de maneira a tratar corretamente essas ocorrências, sem perder apontadores para os nós que ficam na lista.

Como exercício, desenvolva um método que exclua um nó da lista de alunos, cuja chave é uma String passada como parâmetro.

10. Liberação de Elementos Alocados Dinamicamente

Após você ter retirado um elemento da lista, você deverá liberá-lo para que o gerenciador de memória o reutilize quando necessário. Este processo é chamado de "Garbage Collect", o que pode ser traduzido como Coleta de Lixo.

Liberando um nó, que é uma parte da memória, você faz com que o gerenciador de memória coloque este nó numa outra lista ligada, a lista de nós disponíveis, de modo que ele possa recuperá-lo quando houver necessidade de mais memória (pode ser que o seu programa venha a solicitar mais nós, ou, se você estiver num ambiente multi-usuário, outros programas podem solicitar memória, de modo que o gerenciador aloca o espaço liberado após a solicitação).

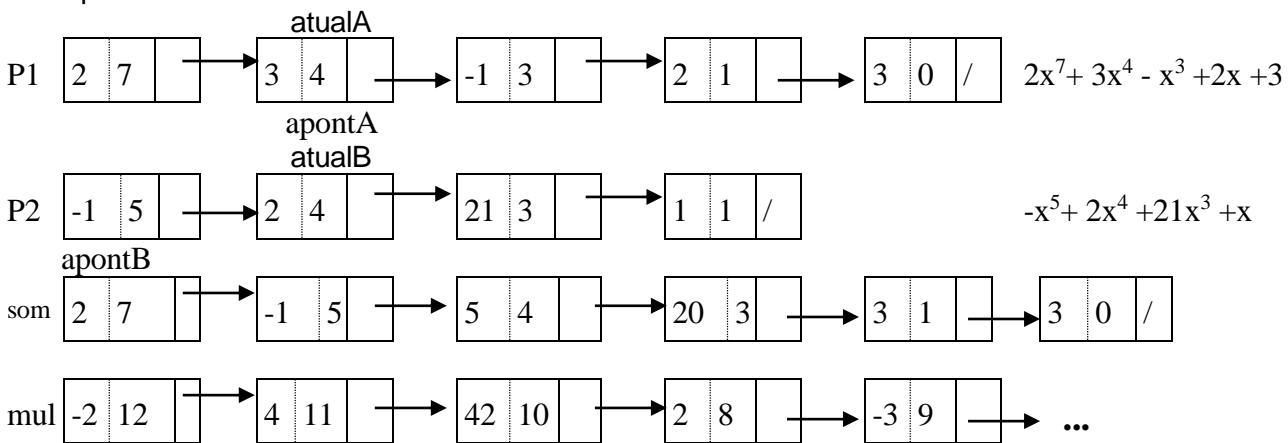
Em Java, basta deixar de utilizar um endereço que ele é automaticamente devolvido para o HEAP. Dessa maneira, a linguagem Java, ao contrário de outras linguagens, possui um mecanismo de coleta de lixo automático.

11. Exercícios

1. Escreva um método que ordene uma lista ligada de alunos, através do processo descrito na seção anterior.
2. Escreva um método que receba como parâmetros os apontadores iniciais de duas listas ligadas, que representam, cada uma, um polinômio de ordem 1 (ou seja, tem apenas uma variável). O nó de cada lista terá um objeto da classe Termo, cuja descrição segue abaixo:

Coeficiente : double
Expoente : byte (somente positivos)

Cada nó armazena um termo do polinômio representado pela lista ligada onde o nó se encontra. As listas estão ordenadas por ordem decrescente do expoente. O método deve calcular e gerar a lista ligada resultante da adição entre o primeiro e o segundo polinômios, devolvendo a lista resultante. Exemplo:

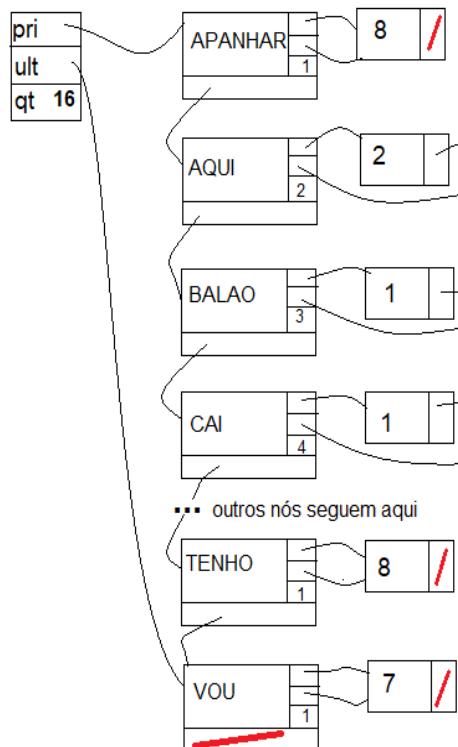


Para realizar a operação de soma, deve-se usar o método de casamento de dados, usando o expoente como critério de ordenação, com um ponteiro para cada lista a ser somada. Esse ponteiro avançará sempre que o expoente apontado for menor que o expoente apontado pelo outro ponteiro. Nesse caso, os dados do nó com o menor expoente são colocados na lista resultante e o processo se repete. Quando os dois expoentes são iguais, faz-se a soma dos coeficientes e esse novo termo é colocado na lista resultante, e os dois ponteiros avançam. Quando uma das listas é percorrida totalmente, os dados da outras listas são copiados na lista resultante até que se percorra também a lista que ainda não foi percorrida totalmente.

3. É dado um arquivo texto contendo palavras (sequências de uma ou mais letras), dígitos e sinais de pontuação. Deve-se produzir um arquivo texto de saída com as seguintes características:

- liste todas as palavras que aparecem no texto, em ordem alfabética; cada palavra só pode aparecer uma vez na lista de saída;
- para cada palavra dada, indicar o número das linhas do texto nas quais ela apareceu. As linhas são numeradas implicitamente de 1 a n, de acordo com a ordem de leitura das linhas do arquivo.
- Mesmo que uma palavra apareça mais de uma vez na mesma linha, ela deve ser listada somente uma vez na saída..

Deve-se usar uma lista ligada para armazenar as diferentes palavras; cada palavra está associada a uma lista ligada com os números de linha onde cada palavra aparece, como se vê na figura abaixo. Em outras palavras, o campo info do nó da lista principal armazenará uma string (a palavra) e uma `ListaSimples<int>`, para relacionar as linhas onde a palavra aparece no arquivo texto.



Arquivo de entrada

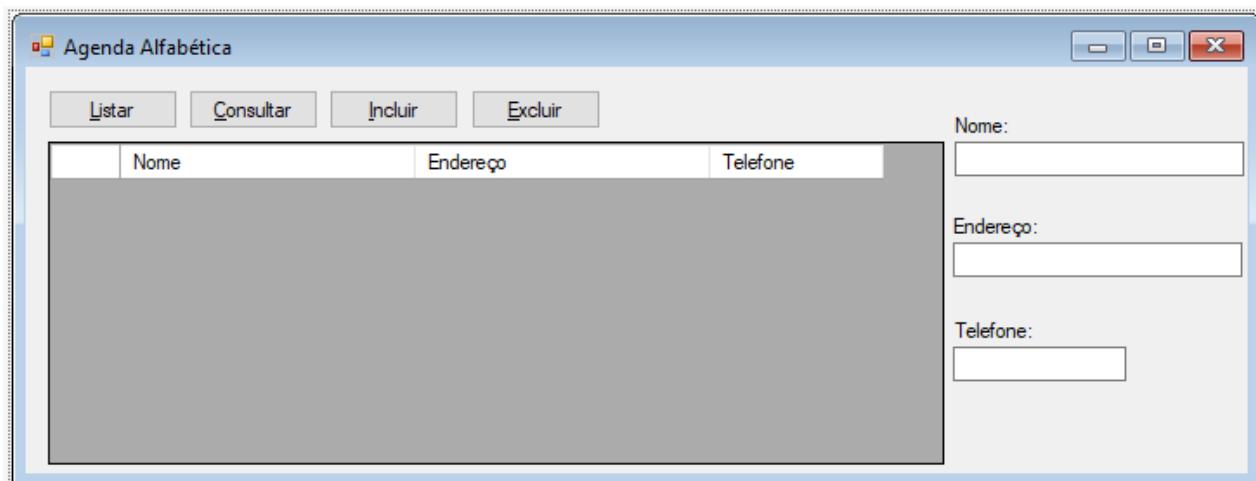
Cai cai balao, cai cai balao
Aqui na minha mao
Nao cai nao, nao cai nao, nao cai
nao
Cai na rua do sabao
Cai cai balao, cai cai balao
Aqui na minha mao
Nao vou la, nao vou la, nao vou la
Tenho medo de apanhar!

Lista de saída

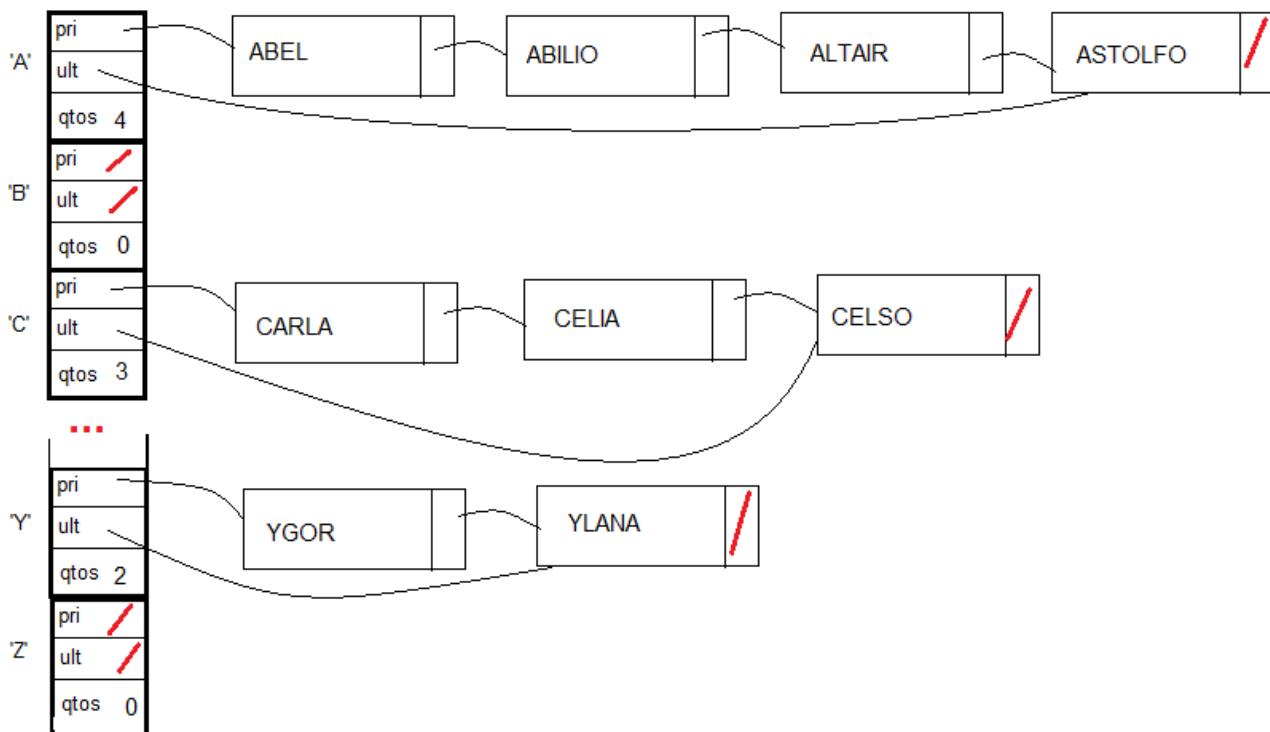
APANHAR:8
AQUI:2 6
BALAO:1 5
CAI:1 3 4 5
DE:8
DO:4
LA:7
MAO:2 6
MEDO:8
MINHA:2 6
NA:2 4 6
NAO:3 7
RUA:4
SABAO:4
TENHO:8
VOU:7

4.(2/87) Faça um programa Windows Forms em C# que monte uma lista ligada ordenada por nome, conforme o esquema abaixo descrito, e que permita operações de consulta, inclusão e exclusão de nós.

Dados da lista : Nome : 30 posições
 Endereço : 30 posições
 Telefone : 15 posições



O acesso é feito pela primeira letra do nome. Teremos um vetor de listas ligadas, onde cada posição contém um objeto ListaSimples que aponta o início e o fim da lista associada à letra inicial, que serve como índice do vetor.



5. Supondo duas listas ligadas de mesma estrutura, ambas ordenadas por um campo chave, contendo também um campo de informações adicionais, faça procedimentos que gerem novas listas ligadas, baseadas nas operações abaixo:
 - a) União de duas listas com essa estrutura
 - b) Diferença entre duas listas com essa estrutura
 - c) Intersecção de duas listas com essa estrutura.
6. Faça um programa em C# que leia um arquivo texto com registros com a descrição a seguir, armazene-os numa lista ligada e efetue as seguintes operações:
 1. Incluir registro
 2. Excluir registro
 3. Listar registros (todos)

Os dados já vêm ordenados pelo campo CHAVE. Não se pode incluir chaves repetidas, nem excluir as não existentes. As opções 1 e 2 são repetitivas, terminando quando CHAVE = 0.

Arquivo: Chave Integer
 Nome String[30]

Ao final do programa, salvar os dados no arquivo original, ordenados. Leve a ordenação em conta nas operações.

4. Um editor de linhas é uma forma de editor de textos que permite a criação e alteração de arquivos texto por meio de comandos escritos numa linha de comando solicitada por um prompt. O primeiro caracter dessa linha de comando indica o código do comando. Em seguida podem vir os parâmetros desse comando. Por exemplo, o comando I indica a inclusão de linhas após a linha atual do texto, como se vê abaixo:

```
*|  
Maria tinha um carneirinho  
O carneirinho de Maria era  
branco como a neve  
#  
*
```

O prompt, no caso, é *. O caracter # indica que terminou a inclusão, quando for o primeiro de uma linha lida. Se você quiser incluir esse caracter no texto, na posição onde ele apareceu no exemplo (caso não fosse o fim do bloco incluído), deveria indicar dois # (## seguido do resto do texto dessa linha).

Faça um programa que implemente esse editor. Note que o conteúdo do arquivo não aparece na tela, a menos que um bloco seja mostrado pelo comando t. Os comandos a implementar são:

A<arquivo>	- abre (ou cria) o <arquivo> indicado
I<texto>...#	- inclui o <texto> após a linha atual
T	- exibe 10 linhas do texto a partir da linha atual
S<cadeia1>@<cadeia2>	- procura, nas linhas a partir da atual, a primeira ocorrência de uma cadeia igual a <cadeia1> e a substitui por <cadeia2>.
B	- vai para primeira linha do arquivo
E	- vai para última linha do arquivo
L	- avança uma linha no arquivo
D<n>	- apaga os n primeiros caracteres da linha atual
K	- apaga a linha atual
G	- salva o arquivo
Q	- sai do programa sem salvar o arquivo

Para implementar as linhas do arquivo, use uma lista ligada de cadeias de até 80 posições. Sempre que uma operação for efetuada, a linha atual fica sendo a última linha visitada pela operação. Lembre-se que, ao avançar a linha atual, pode-se cair numa linha inexistente.

7. (2/93) Um programa que permite a criação de um dicionário de dados simplificado pode ser feito da seguinte maneira:

1. Incluir dados de arquivos
2. Exibir dados
3. Alterar dados
4. Excluir arquivos e seus dados

Um **arquivo** teria as seguintes informações :

Nome do arquivo	String[30]	Nome da chave	String[30]
Nº de campos	byte	Lista de Campos (-)	

Nº de programas Lista de Nomes de Programas em que é usado(-)

Um **campo** teria a seguinte descrição:

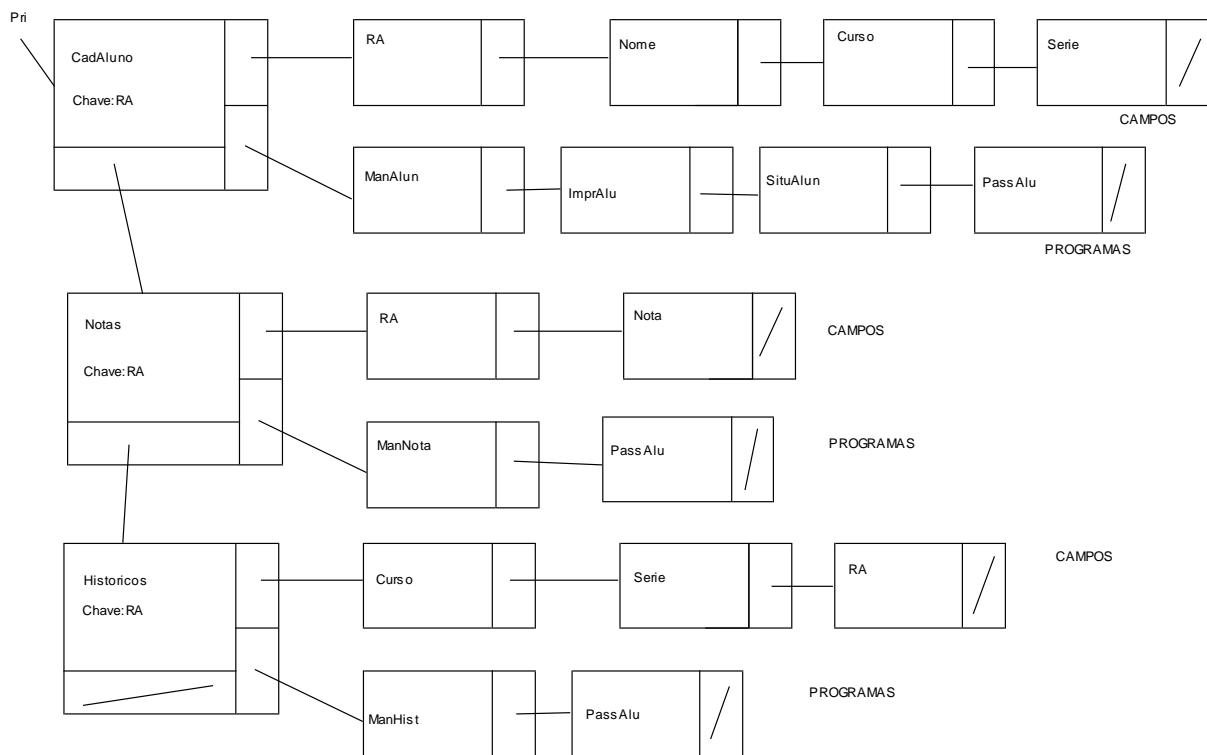
Nome do Campo	String[30]	Tipo do Campo	String[10]
---------------	------------	---------------	------------

O nome de um programa seria uma String[8].

O arquivo que teria de armazenar esta estrutura em disco seria algo como:

Nome do arquivo 1	nº de campos 1	nº de programas 1
Dados do campo 1	... Dados do campo n1	
Programa 1	Programa 2	... Programa m1
Nome do arquivo 2	nº de campos 2	nº de programas 2
Dados do campo 1	... campo n2 (um campo por linha)	
Dados do programa 1	... programa m2 (um programa por linha)	
etc.		

Um possível dicionário poderia ser como abaixo:



Faça um programa em C# que leia um arquivo texto com esse formato e significado, e armazene seus dados nas listas ligadas adequadas, efetuando a seguir as opções do menu, de acordo com as seleções do usuário. Ao final, gravar os dados novamente.

12. Listas Duplamente Ligadas

Uma lista ligada com apenas um apontador para o próximo elemento tem um problema para percorrê-la: isto só pode ser feito num sentido, do primeiro para o último nó.

Para resolver isto, implementou-se uma estrutura de lista com dois apontadores, um para o próximo elemento, e outro para o anterior, de acordo com a figura abaixo:

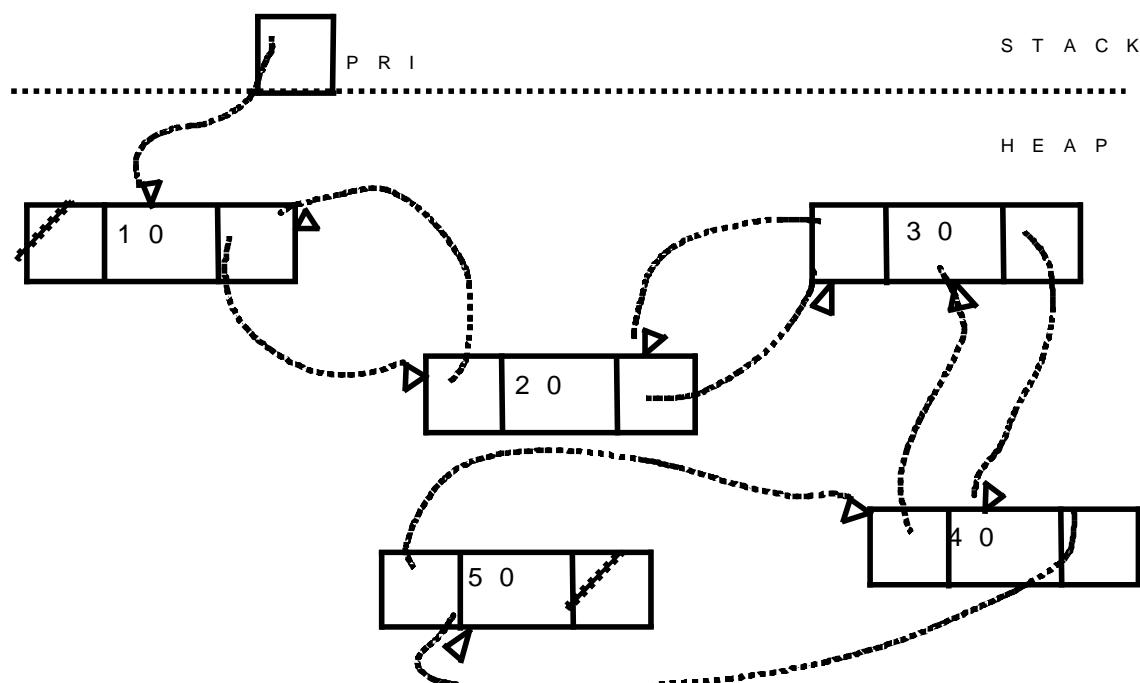
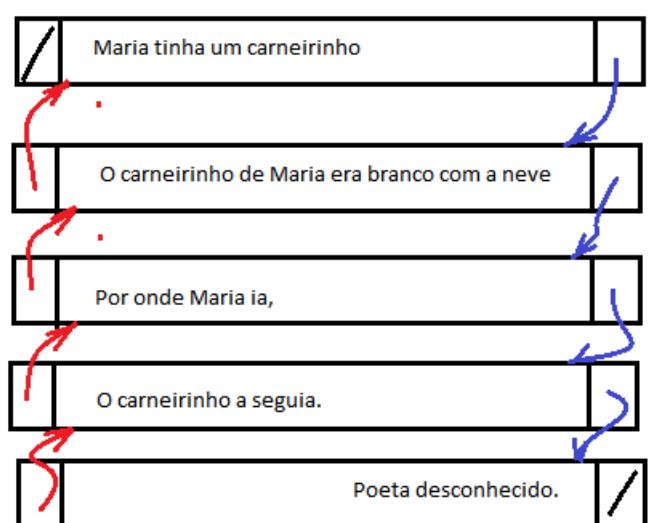


Figura 23 – representação de uma lista duplamente ligada

É aconselhável ter-se, além de um apontador para o início da lista, um apontador para o último nó, para que seja possível percorrê-la nos dois sentidos.



Editor de linhas como lista duplamente ligada

13. Listas Circularmente Ligadas

Ao se criar uma lista ligada, se o último nó for ligado ao primeiro, têm-se uma lista ligada circular. A ideia básica para esta implementação é a simulação de processos repetitivos, de forma circular. Como se define qual é o início, ou qual é o fim da lista? As soluções possíveis são :

- Mantém-se um apontador para o início;
- Não se importar com o início e fim. O início é o nó apontado presentemente, e o fim é encontrado percorrendo-se a lista até se encontrar um nó cujo campo PROXIMO aponte para o nó escolhido como primeiro. Na figura acima, o início seria apontado por P;
- Aponta-se apenas o último nó. A partir dele encontra-se o primeiro.

Com a terceira solução, um único nó apontador permite acesso ao primeiro e último nós da lista circular, e a partir destes temos acesso aos demais.

A figura a seguir ilustra uma lista circularmente ligada, ou lista circular, para simplificar:

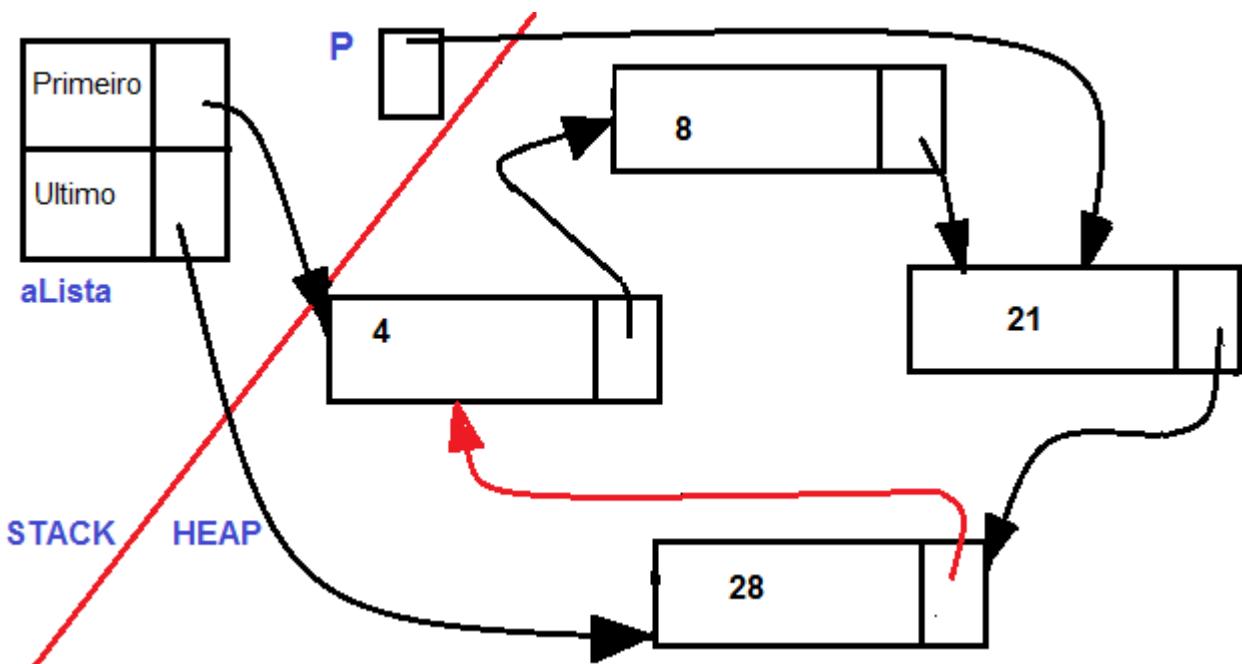


Figura 24 – representação de uma lista circular

14. Cabeças de Listas Ligadas

Cabeças de listas são nós que servem para delimitar regiões específicas de listas ligadas, ou outras informações úteis sobre a estrutura e/ou conteúdo das mesmas.

Geralmente, este nó é o primeiro da lista, e aponta para o nó que realmente mantém o primeiro elemento de dados da lista. O nó de cabeça indicará alguma informação sobre a lista que o segue. Como aplicações para cabeças de listas, temos:

- 1) guardar o número de elementos da lista

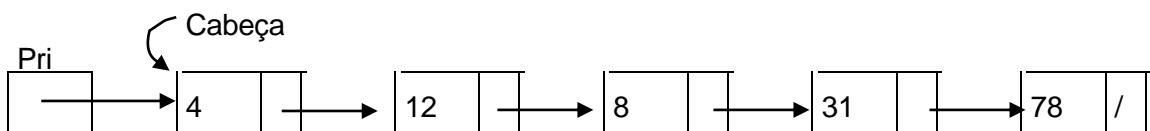


Figura 25 – nó cabeça de lista informando o número de nós da lista ligada

O primeiro nó acima, que é a cabeça da lista, indica quantos nós existem na lista ligada que o segue.

2) relação de classificação

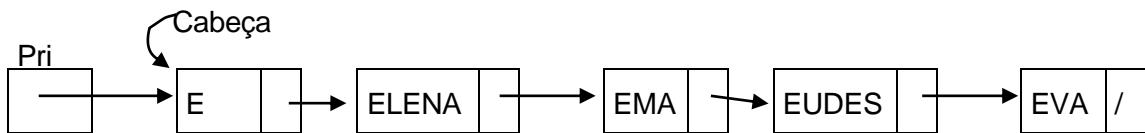


Figura 26 – nó cabeça de lista informando a letra inicial das chaves de classificação dos nós da lista

3) apontador para o final da lista

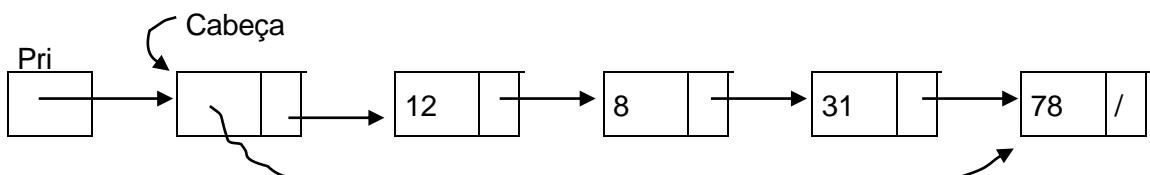


Figura 27 – nó cabeça de lista indicando o último nó da lista ligada

Note que neste caso, o campo de informação do nó cabeça de lista tem um tipo diferente do campo de informação dos demais nós. No nó cabeça, o campo de informação é um apontador do nó de dados, e neste, o campo de informação é um número inteiro. Isso deve ser previsto no momento da declaração dos protótipos (registros) dos nós.

Existem muitos outros usos, que se apresentam de acordo com o tipo de problema a resolver. Os nós de cabeça de lista têm função específica, portanto não devem ter seu conteúdo alterado sem que isso tenha significado correto para sua função.

Geralmente, o nó cabeça é do mesmo tipo do nó usado na lista ligada, mas, dependendo da necessidade, pode ser de tipo diferente, desde que tenha um campo apontando para a lista ligada, como ocorre no terceiro exemplo.

15. Exercícios

1. Usando como modelo a classe `ListaSimples<Dado>`, codigue a classe `ListaDupla<Dado>` com métodos para:

- Incluir um Dado antes do primeiro nó;
- Incluir um Dado após o último nó;
- Incluir um nó, em ordem;
- Buscar o nó que armazenado o Dado passado como parâmetro
- Excluir um nó;
- Ordenar a lista em ordem crescente de acordo com o critério de `CompareTo(Dado)`;
- Retornar um `List<Dado>` com os elementos da `ListaDupla` em ordem crescente;
- Retornar um `List<Dado>` com os elementos da `ListaDupla` em ordem decrescente.

2. Desenvolva em C# a classe `ListaCircular<Dado>`, com métodos para ordenar, incluir em ordem, incluir antes do primeiro, incluir depois do último, incluir em ordem crescente do campo de comparação, procurar elementos e excluir um dado a ser procurado. Parta do princípio de que essa lista circular terá um apontador para o último nó, apenas.

3. Desenvolva em C# a classe de lista duplamente ligada circular que tenha um apontador para o nó inicial.



5. Na classe ListaCircular codifique um método que recebe, como parâmetro, uma instância de ListaCircular e a una com a ListaCircular this, supondo que as duas listas circulares estão ordenadas crescentemente e gerando uma terceira lista ligada circular ordenada que será retornada como resultado do método. Use a ideia de casamento de listas ordenadas.
6. Existe uma guarnição da polícia do Rio de Janeiro no morro da Rocinha; tal grupamento está cercado por traficantes e não tem munição para mais de duas horas, nem mesmo meios de comunicação à distância para pedirem reforços (por exemplo, rádios, fumaça, telepatia);

Precisam, assim, escolher alguém que consiga passar pelo cerco e encontrar o departamento de polícia para pedir que enviem socorro.

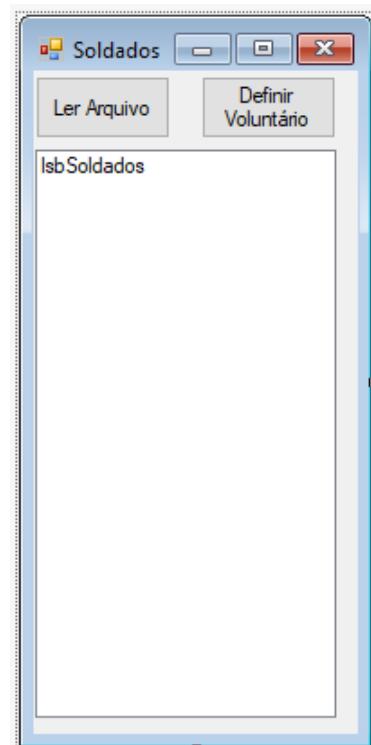
Para escolher o "voluntário", o chefe da guarnição, também conhecido por Toninho Malvadeza, "bolou" o seguinte esquema:

- Perfilou os soldados num círculo, com ele no centro (esperto);
- Assinalou o soldado com o menor número de matrícula;
- Contou, a partir deste soldado, da esquerda para a direita, tantos soldados quanto o número de matrícula do primeiro soldado;
- No último soldado contado, anotou o número de matrícula do mesmo e contou, da direita para a esquerda, tantos soldados quanto este segundo número;
- O soldado em que terminou a contagem foi retirado do círculo;
- Recomeçou a contagem a partir do soldado seguinte (da esquerda para a direita) ao que foi retirado;

Este processo foi repetido até que apenas um soldado sobrasse do círculo original. Este foi o "feliz" escolhido para a missão.

Crie um programa Windows Forms em C#, que simule tal processo, tendo como entrada um arquivo com o nome e o número de matrícula de cada soldado.

Pode-se usar o formulário da figura ao lado como modelo para essa aplicação. No Listbox, deve-se mostrar, a cada rodada, como está a lista circular e, ao final, o soldado selecionado como "voluntário".



4. Hashing

Hashing ou **espalhamento** é uma técnica muito comum para armazenar dados de uma maneira que possam ser inseridos e recuperados muito rapidamente.

Hashing usa uma estrutura de dados chamada **tabela de hash**. Embora tabelas de hash forneçam inserção, remoção e recuperação rápidas, operações que envolvam pesquisa, tais como encontrar os valores máximo e mínimo, não são feitas muito rapidamente.

Para esses tipos de operações, outras estruturas de dados, como as árvores de busca, são preferidas.

A biblioteca de classes do .NET Framework fornece uma classe muito útil para trabalhar com tabelas de hash, a classe Hashtable. Discutiremos essa classe neste capítulo, mas também discutiremos como implementar nossa própria tabela de hash. Construir tabelas de hash não é muito difícil e as técnicas de programação usadas são bastante válidas de se conhecer.

1. Como Hashing Funciona

Uma estrutura de dados para tabela de hash é projetada ao redor de um vetor. O vetor consiste de elementos 0 até algum tamanho pré-determinado, embora possamos aumentar o tamanho posteriormente se necessário. Cada item de dados é armazenado no vetor baseado em alguma parte dos dados, chamada a *chave*. Para armazenar um elemento na tabela de hash, a chave é mapeada em um número no intervalo de 0 até o tamanho da tabela de hash usando uma função chamada uma função de hash.

O objetivo ideal da função de hash é armazenar cada chave em sua própria posição no vetor. No entanto, devido ao fato de haver um número ilimitado de chaves possíveis e um número finito de posições no vetor, um objetivo mais realista da função de hash é tentar distribuir as chaves da forma mais uniforme (espalhada) possível dentre as posições do vetor.

Mesmo com uma boa função de hash, é possível que, para duas chaves diferentes, obter o mesmo valor da função e, por conseguinte, a função definir a mesma posição de armazenamento para duas chaves diferentes. Isto é chamado de **colisão** e temos de definir uma estratégia para lidar com colisões quando elas ocorrem.

A última coisa que temos de definir é quão grande deve ser a dimensão do vetor usado como tabela de hash. Primeiramente, é recomendado que o tamanho do vetor seja um número primo. Explicaremos o motivo dessa recomendação quando examinarmos as diferentes funções de hash. Há várias estratégias diferentes para determinar o tamanho adequado do vetor, todas elas baseadas na técnica usada para lidar com colisões.

2. Escolhendo uma Função de HASH

A escolha de uma função de hash depende do tipo de dados da chave que você está usando para identificar os dados. Se sua chave é um inteiro, a função mais simples é retornar o resto da divisão da chave pelo tamanho do vetor, o que é chamado de **método da divisão**:

$$\text{posição de hash} = \text{chave \% número de posições na tabela}$$

Existem circunstâncias em que este método não é recomendado, como quando todas as chaves terminam em zero e o tamanho do vetor é 10. Esta é uma das razões pelas quais o tamanho do vetor sempre deve ser um número primo. Além disso, se as chaves são números inteiros aleatórios então a função de hash deveria distribuí-las mais uniformemente.

Em muitas aplicações, no entanto, as chaves são strings. A escolha de uma função de hash para trabalhar com chaves string é mais difícil e essa funções devem ser escolhidas cuidadosamente. Uma função simples que à primeira vista aparenta funcionar bem é somar os valores ASCII de cada caracter da chave. O valor da posição na tabela de hash (ou valor de hash, para abreviar) seria o resto da divisão entre essa soma pelo tamanho do vetor.

A classe abaixo mostra uma primeira versão de uma tabela de Hash, com função de hash como a citada acima:

```
class HashSimples
{
    const int tamanhoPadrao = 10007;
    string[] tabelaDeHash;

    public HashSimples() : this(tamanhoPadrao) { }

    public HashSimples(int tamanhoDesejado)
    {
        tabelaDeHash = new string[tamanhoDesejado];
    }

    private int Hash(string chave)
    {
        int tot = 0;
        for (int i = 0; i < chave.Length; i++)
            tot += (int)chave[i];
        return tot % tabelaDeHash.Length;
    }

    public string Incluir(string chave)
    {
        string saida = "";
        int valorDeHash = Hash(chave.Trim()); // posicao calculada para um registro
        if (tabelaDeHash[valorDeHash] != null) // ja há dado armazenado nessa posição
            saida = $"colisao na posicao {valorDeHash} entre " +
                    $"{tabelaDeHash[valorDeHash]} e {chave}";
        tabelaDeHash[valorDeHash] = chave;
        return saida;
    }

    public bool Existe(string s, out int posicao)
    {
        posicao = Hash(s);
        return tabelaDeHash[posicao] == s;
    }

    public List<string> Conteudo()
    {
        var saida = new List<string>();
        for (int i = 0; i < tabelaDeHash.Length; i++)
            if (tabelaDeHash[i] != null)
                saida.Add($"{i}, {tabelaDeHash[i]}");
            //else
            //    saida.Add($"{i} ");
        return saida;
    }

    public void Limpar()
    {
        for (int i = 0; i < tabelaDeHash.Length; i++)
            tabelaDeHash[i] = null;
    }
}
```

Construtores da classe. Caso seja chamado sem parâmetro, o primeiro construtor (default) chama o segundo construtor passando o tamanho padrão (um primo) como parâmetro

Esta função soma o código ASCII de cada caracter da chave, calcula o resto da divisão desta soma pelo tamanho do vetor e retorna esse resultado como sendo o índice onde o elemento deverá ser armazenado (ou onde deveria estar)

Este método calcula a posição em que a chave deveria ser armazenada, usando a função Hash().
Se já houver valor armazenado nessa posição, retornará uma mensagem informando a colisão.
Armazena a nova chave na posição calculada

Este método calcula a posição em que uma chave deveria estar na tabela de Hash e informa se ela está ou não armazenada nessa posição.

Este método retorna uma lista de strings com o conteúdo da tabela de hash

Este método limpa todos os elementos da tabela para reutilizá-la

O programa (modo Console) abaixo mostra como essa classe e seus métodos se comportam:

```
using System;
using static System.Console;

namespace apHashSimples
{
    class Program
    {
        static HashSimples tabela1;
        static string[] algunsNomes = new string[]
            {"David", "Jennifer", "Donnie", "Mayo", "Raymond",
             "Bernica", "Mike", "Clayton", "Beata", "Michael",
             "Felipe", "Silvana", "Igor", "Lucia", "Guilherme",
             "Monica"} //,"Amélia"};

        static void Main(string[] args)
        {
            int onde;
            string resultado;

            ForegroundColor = ConsoleColor.Black;
            BackgroundColor = ConsoleColor.White;
            Clear();
            WriteLine("Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.");
            WriteLine($"Há {algunsNomes.Length} nomes no conjunto de testes.\nVocê pode "+
                    "digitar um valor menor que esse, forçando colisões.");

            Write("\nQual o número de posições da Tabela de Hash?");
            int tamanho = int.Parse(ReadLine());
            tabela1 = new HashSimples(tamanho);

            WriteLine("\n-----");
            WriteLine("Teste com hash inicial, sem uso da regra de Horner no cálculo");

            for (int i = 0; i < algunsNomes.Length; i++)
                if ((resultado = tabela1.Incluir(algunsNomes[i])) != "")
                    WriteLine(resultado);

            WriteLine("Conteúdo da tabela");
            foreach (string item in tabela1.Conteudo())
                WriteLine(item);
            EsperarEnter();

            if (tabela1.Existe("Amélia", out onde))
                WriteLine($"Achou Amélia na posição {onde}");
            else
                WriteLine($"Não achou Amélia mas deveria estar na posição {onde}");

            if (tabela1.Existe("Raymond", out onde))
                WriteLine($"Achou Raymond na posição {onde}");
            else
                WriteLine($"Não achou Raymond mas deveria estar na posição {onde}");

            EsperarEnter();
        }

        static void EsperarEnter()
        {
            Write("\nPressione [Enter]");
            ReadLine();
        }
    }
}
```

A seguir, temos alguns testes desse programa, com vários tamanhos de tabela de Hash:

- a. Com tabela com tamanho igual a 16 elementos: ocorrem várias colisões e, portanto, perda de dados. Mude o código para incluir, também, Amélia na relação original de dados e verá que ela colide com Beata:

```
D:\disciplinas\2oSemestre\Estruturas de Dados 2\Classes\Hash2020\apHash...
Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.
Há 16 nomes no conjunto de testes.
Você pode digitar um valor menor que esse, forçando colisões.

Qual o número de posições da Tabela de Hash?16

-----
Teste com hash inicial, sem uso da regra de Horner no cálculo
colisao na posicao 6 entre Mayo e Mike
colisao na posicao 10 entre Raymond e Clayton
colisao na posicao 13 entre Donnie e Beata
colisao na posicao 1 entre Jennifer e Igor
colisao na posicao 14 entre Silvana e Lucia
Conteúdo da tabela
 1 : Igor
 2 : Guilherme
 3 : Michael
 4 : Bernica
 5 : Felipe
 6 : Mike
 7 : Monica
 8 : David
10 : Clayton
13 : Beata
14 : Lucia

Pressione [Enter]
Não achou Amélia mas deveria estar na posição 13
Não achou Raymond mas deveria estar na posição 10

Pressione [Enter]..
```

```
Selecionar D:\disciplinas\2oSemestre\Estruturas de Dados 2\Classes\Hash2020...
Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.
Há 17 nomes no conjunto de testes.
Você pode digitar um valor menor que esse, forçando colisões.

Qual o número de posições da Tabela de Hash?16

-----
Teste com hash inicial, sem uso da regra de Horner no cálculo
colisao na posicao 6 entre Mayo e Mike
colisao na posicao 10 entre Raymond e Clayton
colisao na posicao 13 entre Donnie e Beata
colisao na posicao 1 entre Jennifer e Igor
colisao na posicao 14 entre Silvana e Lucia
colisao na posicao 13 entre Beata e Amélia
Conteúdo da tabela
 1 : Igor
 2 : Guilherme
 3 : Michael
 4 : Bernica
 5 : Felipe
 6 : Mike
 7 : Monica
 8 : David
10 : Clayton
13 : Amélia
14 : Lucia

Pressione [Enter]
Achou Amélia na posição 13
Não achou Raymond mas deveria estar na posição 10

Pressione [Enter]
```

- b. Com tabela com tamanho igual a 113 elementos: ocorrem menos colisões e, portanto, menor perda de dados. Já com 10007 elementos ocorreu apenas uma colisão. Mesmo com 100007 elementos, ainda haverá uma colisão.

```
D:\disciplinas\2oSemestre\Estruturas de Dados 2\Classes\Hash2020\apHash...
Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.
Há 16 nomes no conjunto de testes.
Você pode digitar um valor menor que esse, forçando colisões.

Qual o número de posições da Tabela de Hash?113

-----
Teste com hash inicial, sem uso da regra de Horner no cálculo
colisao na posicao 52 entre Raymond e Clayton
colisao na posicao 40 entre Donnie e Silvana
colisao na posicao 26 entre Jennifer e Guilherme
Conteúdo da tabela
 13 : Michael
 14 : Bernica
 25 : Beata
 26 : Guilherme
 32 : Felipe
 34 : Monica
 36 : David
 40 : Silvana
 42 : Lucia
 51 : Mike
 52 : Clayton
 62 : Igor
 67 : Mayo

Pressione [Enter]
Não achou Amélia mas deveria estar na posição 39
Não achou Raymond mas deveria estar na posição 52

Pressione [Enter]..
```

```
D:\disciplinas\2oSemestre\Estruturas de Dados 2\Classes\Hash2020\apHash...
Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.
Há 16 nomes no conjunto de testes.
Você pode digitar um valor menor que esse, forçando colisões.

Qual o número de posições da Tabela de Hash?10007

-----
Teste com hash inicial, sem uso da regra de Horner no cálculo
colisao na posicao 730 entre Raymond e Clayton
Conteúdo da tabela
 390 : Mike
 401 : Igor
 406 : Mayo
 477 : Beata
 488 : David
 494 : Lucia
 597 : Felipe
 599 : Monica
 605 : Donnie
 691 : Michael
 692 : Bernica
 718 : Silvana
 730 : Clayton
 817 : Jennifer
 930 : Guilherme

Pressione [Enter]
Não achou Amélia mas deveria estar na posição 717
Não achou Raymond mas deveria estar na posição 730

Pressione [Enter]
```

O método Conteudo() nos retorna onde os nomes são efetivamente colocados dentro do vetor pela função de hash. Como podemos ver, a distribuição não é muito uniforme. Os nomes são agrupados no começo e no final do vetor, ou em grupos (clusters) próximos no meio do vetor.

No entanto, há um problema ainda maior escondido aqui. Nem todos os nomes são exibidos, ou seja, nem todos puderam ser armazenados. De forma interessante, se mudarmos o tamanho do vetor para um número primo, diminui o número de colisões e poderá acontecer de até todos os nomes serem armazenados apropriadamente. Portanto, uma regra importante quando você escolher o tamanho de sua tabela de hash (e quando usar uma função de hash semelhante à que estamos usando aqui) é escolher um número primo para o tamanho da tabela.

O tamanho que você, por fim, escolher, dependerá da determinação do número de registros armazenados na tabela de hash, mas um número seguro parece ser 10.007 (dado que você não está realmente desejando armazenar muitos número na sua tabela). O número 10.007 é primo e não é tão grande que exija o uso de muita memória de forma que degrade o desempenho do seu programa.

Apegando-nos à idéia básica de usar o valor ASCII total da chave na criação da tabela de hash, podemos melhor o algoritmo para ainda mais espalhar os valores pela tabela (lembre-se dos agrupamentos acima) e diminuir o número de colisões. O próximo algoritmo aprimora a distribuição no vetor e, por isso, o colocaremos em uma nova classe, chamada HashAprimorado. Primeiramente, observemos o código, seguido de uma explicação:

```
static int Hash (string chave)
{
    long tot = 0;
    for (int i = 0; i < chave.Length; i++)
        tot += 37 * tot + (int)chave[i];
    tot = tot % tabelaDeHash.Length;
    if (tot < 0)
        tot += tabelaDeHash.Length;
    return (int)tot;
}
```

Esta função usa a **regra de Horner para calcular a função polinomial de 37**. Ela é um método numérico que permite espalhar melhor os valores resultantes.

Observemos nas figuras abaixo a distribuição das chaves usando esta nova função, com 131 posições no vetor. Não houve colisões. Mesmo incluindo mais um registro (Amélia) não houve colisões.

A classe HashAprimorado tem o código abaixo. Note que praticamente são os mesmos métodos que a classe HashSimples. Apenas alteramos a função de Hash, com o método numérico citado acima.

```
class HashAprimorado
{
    const int tamanhoPadrao = 10007;
    string[] tabelaDeHash;

    public HashAprimorado() : this(tamanhoPadrao) { }

    public HashAprimorado(int tamanhoDesejado)
    {
        tabelaDeHash = new string[tamanhoDesejado];
    }

    private int Hash(string chave)
    ... // mesmo código anteriormente visto

    public string Incluir(string chave)
    {
        string saida = "";
        int valorDeHash = Hash(chave.Trim()); // posicao calculada para um registro
        if (tabelaDeHash[valorDeHash] != null)
            saida = $"colisao na posicao {valorDeHash} entre " +
                    $"{tabelaDeHash[valorDeHash]} e {chave}";
        tabelaDeHash[valorDeHash] = chave;
        return saida;
    }

    public bool Existe(string s, out int posicao)
    {
        posicao = Hash(s);
        if (tabelaDeHash[posicao] == s)
            return true;
        else
            return false;
    }

    public List<string> Conteudo()
    {
        var saida = new List<string>();
        for (int i = 0; i < tabelaDeHash.Length; i++)
            if (tabelaDeHash[i] != null)
                saida.Add($"{i},5 : {tabelaDeHash[i]}");
        //else
        //    saida.Add($"{i} ");
    }

    public void Limpar()
    {
        for (int i = 0; i < tabelaDeHash.Length; i++)
            tabelaDeHash[i] = null;
    }
}
```

O programa que usa essa classe também é praticamente o mesmo que antes, apenas mudamos da classe HashSimples para a classe HashAprimorado e mudamos algumas mensagens.

```
class Program
```

```
{  
    static HashAprimorado tabela;  
  
    static string[] algunsNomes = new string[]  
    {"David", "Jennifer", "Donnie", "Mayo", "Raymond",  
     "Bernica", "Mike", "Clayton", "Beata", "Michael",  
     "Felipe", "Silvana", "Igor", "Lucia", "Guilherme",  
     "Monica", "Amélia"};  
    static void Main(string[] args)  
{  
        int onde;  
        string resultado;  
  
        BackgroundColor = ConsoleColor.White;  
        ForegroundColor = ConsoleColor.Black;  
        Clear();  
        WriteLine("Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.");  
        WriteLine($"Há {algunsNomes.Length} nomes no conjunto de testes.\nVocê pode  
digitar um valor menor que esse, forçando colisões.");  
  
        Write("\nQual o número de posições da Tabela de Hash?");  
        int tamanho = int.Parse(ReadLine());  
  
        WriteLine("\n-----");  
        WriteLine("Teste com hash aprimorado, usando a regra de Horner no cálculo");  
  
        tabela = new HashAprimorado(tamanho);  
  
        for (int i = 0; i < algunsNomes.Length; i++)  
            if ((resultado = tabela.Incluir(algunsNomes[i])) != "")  
                WriteLine(resultado);  
  
        WriteLine("Conteúdo da tabela");  
        foreach (string item in tabela.Conteudo())  
            WriteLine(item);  
  
        if (tabela.Existe("Amélia", out onde))  
            WriteLine($"Achou Amélia na posição {onde}");  
        else  
            WriteLine($"Não achou Amélia mas deveria estar na posição {onde}");  
  
        if (tabela.Existe("Raymond", out onde))  
            WriteLine($"Achou Raymond na posição {onde}");  
        else  
            WriteLine($"Não achou Raymond mas deveria estar na posição {onde}");  
  
        EsperarEnter();  
    }  
    static void EsperarEnter()  
{  
        Write("\nPressione [Enter]");  
        ReadLine();  
    }  
}
```

Agora, com vetor de 10007 posições:

The image shows two side-by-side windows of a Windows application. Both windows have a title bar with the path 'D:\disciplinas\2oSemestre\Estruturas de Dados 2\Classes\Hash2020\apHashA...' and a close button 'X'. The left window displays a message: 'Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.' followed by 'Há 17 nomes no conjunto de testes.' and 'Você pode digitar um valor menor que esse, forçando colisões.' Below this, it asks 'Qual o número de posições da Tabela de Hash?10007' and shows a list of names with their corresponding hash values. The right window shows a similar message and asks 'Qual o número de posições da Tabela de Hash?100'. It also lists names with their hash values. Both windows show a message at the bottom: 'Pressione [Enter]'.

Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.
Há 17 nomes no conjunto de testes.
Você pode digitar um valor menor que esse, forçando colisões.

Qual o número de posições da Tabela de Hash?10007

Teste com hash aprimorado, usando a regra de Horner no cálculo
Conteúdo da tabela

241 : Amélia
1272 : Felipe
1540 : Silvana
1959 : Lucia
2566 : Guilherme
3288 : David
3646 : Bernica
5181 : Donnie
5381 : Beata
5414 : Monica
5750 : Jennifer
5815 : Igor
6869 : Mayo
7872 : Mike
7913 : Raymond
9043 : Clayton
9230 : Michael

Achou Amélia na posição 241
Achou Raymond na posição 7913

Pressione [Enter]

Selecionar D:\disciplinas\2oSemestre\Estruturas de Dados 2\Classes\Hash2020...

Teste com tamanhos primo e não primo. Exemplo: 100, 131, 10007.
Há 17 nomes no conjunto de testes.
Você pode digitar um valor menor que esse, forçando colisões.

Qual o número de posições da Tabela de Hash?100

Teste com hash aprimorado, usando a regra de Horner no cálculo
colisao na posicao 21 entre Mayo e Beata
colisao na posicao 31 entre Mike e Monica
Conteúdo da tabela

3 : Lucia
8 : Clayton
14 : David
15 : Bernica
19 : Guilherme
20 : Igor
21 : Beata
31 : Monica
32 : Jennifer
41 : Silvana
49 : Felipe
52 : Raymond
59 : Amélia
71 : Donnie
74 : Michael

Achou Amélia na posição 59
Achou Raymond na posição 52

Pressione [Enter]

Nas duas figuras acima podemos perceber que os dados estão distribuídos mais uniformemente, embora seja difícil verificar isso com certeza usando um conjunto de dados tão pequeno. Notamos, porém, que o vetor com 100 posições ainda deixa de armazenar um dos dados. Isso é devido a uma colisão, que ocorreu no elemento 37.

A vantagem de usar hash é que, com um gasto adicional de memória, você consegue recuperar dados em tempo $O(1)$, pois não há acesso sequencial ou binário dos dados, e sim acesso direto (claro que se perde um pequeno tempo calculando o valor de hash).

3. Pesquisando dados numa Tabela de Hash

Para buscar dados na tabela de hash, precisamos calcular o valor de hash para a chave desejada e então acessar o elemento do vetor. Eis a função:

```
public bool Existe(string s, out int posicao)
{
    posicao = Hash(s);
    return tabelaDeHash[posicao] == s;
}
```

Esta função retorna True se o item está na tabela de Hash e falso caso contrário.

Não precisamos comparar o tempo gasto por esta função com o tempo de pesquisa sequencial ou binária do vetor, uma vez que esta função claramente executa em menos tempo, a menos que o dado procurado esteja no início do vetor (na pesquisa sequencial) ou exatamente na posição média (na pesquisa binária).

Mas, e se houve colisões e o dado procurado não se encontra na tabela, pois foi substituído pelo que colidiu com ele?

4. Lidando com Colisões

Quando trabalhamos com tabelas de hash, é **inevitável** que encontremos situações onde o valor de hash para uma chave acabe resultando em um valor que já foi usado para armazenar outra chave. Isto é chamado de **colisão** e há muitas técnicas que você pode usar quando uma colisão ocorre. Estas técnicas incluem bucket hashing, endereçamento aberto e hashing duplo (entre outras). Vamos cobrir brevemente cada uma dessas técnicas.

Bucket Hashing

Quando originalmente definimos o que é uma tabela de hash, afirmamos que é preferível que somente um dado resida em uma posição da tabela de hash. Isso funciona otimamente se não existem colisões, mas se uma função de hash retorna o mesmo valor para dois itens de dados diferentes, temos um problema.

Uma solução para o problema da colisão é implementar a tabela de hash usando *buckets* (baldes). Um bucket é, simplesmente, uma estrutura de dados simples, armazenada num elemento da tabela de hash, que permite armazenar múltiplos itens. Na maioria das implementações, esta estrutura de dados é um vetor, mas em nossa implementação vamos usar uma lista indexada (arrayList), o que nos permitirá não nos preocuparmos com ficar sem memória e também permitirá alocar mais espaço quando necessário. No fim, isso fará nossa aplicação mais eficiente.

Para inserir um item, primeiramente usamos a função de hash para determinar em qual posição da tabela de hash armazenaremos o item. Então verificamos se o item já está no arrayList armazenado nessa posição; se estiver não fazemos nada (seria uma inclusão repetida). Se não está, então chamamos o método Add armazenar esse item.

Para remover um item de uma tabela de hash, novamente determinamos, em primeiro lugar, o valor de hash do item a ser removido e vamos para essa posição da tabela. Em seguida verificamos se esse item está no arrayList armazenado nessa posição e, se estiver, o removemos.

Segue o código para uma classe BucketHash que inclui a função de Hash aprimorada e os métodos Inserir(), Remover(), Existe() e Conteudo(). Em seguida, o código do programa principal:

```
using System;
using System.Collections;
using System.Collections.Generic;

class BucketHash
{
    private const int SIZE = 37; // para gerar mais colisões; o ideal é primo > 100
    ArrayList[] dados;

    public BucketHash()
    {
        dados = new ArrayList[SIZE];
        for (int i = 0; i < SIZE; i++)
            dados[i] = new ArrayList(1);
    }

    private int Hash(string chave)
    {
        long tot = 0;

        for (int i = 0; i < chave.Length; i++)
            tot += 37 * tot + (char)chave[i];
```

```

        tot = tot % dados.Length;
        if (tot < 0)
            tot += dados.Length;

        return (int)tot;
    }

    public void Inserir(string item)
    {
        int valorDeHash = Hash(item);
        if (!dados[valorDeHash].Contains(item))
            dados[valorDeHash].Add(item);
    }

    public bool Remover(string item)
    {
        int onde = 0;
        if (!Existe(item, out onde))
            return false;
        dados[onde].Remove(item);
        return true;
    }

    public bool Existe(string chave, out int onde)
    {
        onde = Hash(chave);
        return dados[onde].Contains(chave);
    }

    public List<string> Conteudo()
    {
        List<string> saida = new List<string>();
        for (int i = 0; i < dados.Length; i++)
            if (dados[i].Count > 0)
            {
                string linha = $"{i,5} : ";
                foreach (string chave in dados[i])
                    linha += " | " + chave;
                saida.Add(linha);
            }
        return saida;
    }
}

```

Programa Principal em modo console:

```

using System;
using System.Collections.Generic;
using static System.Console;

namespace apBucketHash
{
    class Program
    {
        static void Main(string[] args)
        {
            BucketHash balde = new BucketHash();

```

```

string[] algunsNomes = new string[]
{
    "David", "Jennifer", "Donnie", "Mayo", "Raymond",
    "Bernica", "Mike", "Clayton", "Beata", "Michael",
    "Felipe", "Silvana", "Michael", "Lucia", "Guilherme",
    "Monica", "Amélia"};
}

BackgroundColor = ConsoleColor.White;
ForegroundColor = ConsoleColor.Black;
Clear();

WriteLine("Inserindo chaves");
for (int i=0; i < algunsNomes.Length; i++)
    balde.Inserir(algunsNomes [i]);
Exibir(balde.Conteudo());
EsperarEnter();

if (balde.Remover("Bernica"))
    WriteLine("Removeu: Bernica");
else
    WriteLine("Não achou: Bernica");
Exibir(balde.Conteudo());
EsperarEnter();

if (balde.Remover("David"))
    WriteLine("Removeu: David");
else
    WriteLine("Não achou: David");
Exibir(balde.Conteudo());
EsperarEnter();

if (balde.Remover("Chico"))
    WriteLine("Removeu: Chico");
else
    WriteLine("Não achou: Chico");
Exibir(balde.Conteudo());
EsperarEnter();
}

static void Exibir(List<string> lista)
{
    foreach (string item in lista)
        WriteLine(item);
}

static void EsperarEnter()
{
    Write("Pressione [Enter]:");
    ReadLine();
    WriteLine();
}
}

```

```

D:\disciplinas\2oSemestre\Est...  —  □  X
Inserindo chaves
  3 : | Jennifer
  5 : Felipe | Guilherme
  7 : David | Monica
 13 : Donnie | Lucia
 14 : Amélia
 15 : Silvana
 20 : Mike
 25 : Michael
 26 : Bernica
 27 : Raymond | Clayton
 33 : Beata
 36 : Mayo
Pressione [Enter]: 

Removeu: Bernica
  3 : | Jennifer
  5 : Felipe | Guilherme
  7 : David | Monica
 13 : Donnie | Lucia
 14 : Amélia
 15 : Silvana
 20 : Mike
 25 : Michael
 27 : Raymond | Clayton
 33 : Beata
 36 : Mayo
Pressione [Enter]: 

Removeu: David
  3 : | Jennifer
  5 : Felipe | Guilherme
  7 : Monica
 13 : Donnie | Lucia
 14 : Amélia
 15 : Silvana
 20 : Mike
 25 : Michael
 27 : Raymond | Clayton
 33 : Beata
 36 : Mayo
Pressione [Enter]: 

```

Quando usamos bucket hashing, a coisa mais importante que devemos fazer é manter o número de elementos usados no arraylist o mais baixo possível. Isto minimiza o trabalho extra que deve ser feito quando da inclusão ou remoção de itens da tabela de hash.

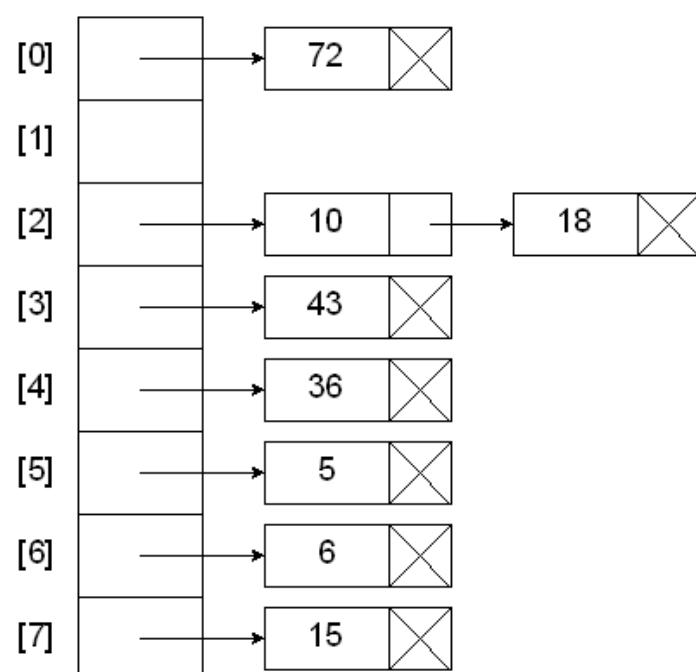
No código acima, minimizamos o tamanho do arraylist ao configurar a capacidade inicial de cada arraylist em 1, na chamada do construtor. Quando ocorrer colisão, a capacidade do arraylist é mudada para 2, e então a capacidade continua a dobrar a cada vez que esse arraylist fica cheio.

Com uma boa função de hash, no entanto, o arraylist não deverá ficar muito grande, pois uma boa função de hash faz bom espalhamento.

A razão entre o número de elementos na tabela de hash e o tamanho da tabela é chamado de *fator de carga*. Estudos demonstraram que o desempenho da tabela de hash é melhor quando o fator de carga é 1.0, ou seja, quando o tamanho da tabela é exatamente igual ao número de elementos armazenados.

Hash key = key % table size

$$\begin{array}{ll} 4 & = 36 \% 8 \\ 2 & = 18 \% 8 \\ 0 & = 72 \% 8 \\ 3 & = 43 \% 8 \\ 6 & = 6 \% 8 \\ 2 & = 10 \% 8 \\ 5 & = 5 \% 8 \\ 7 & = 15 \% 8 \end{array}$$



Na figura acima, exibimos o bucket hash com uma função de hash usando o método da divisão
(fonte: <http://faculty.cs.niu.edu/~freedman/340/340notes/340hash.htm>)

Endereçamento Aberto

Encadeamento separado diminui o desempenho da sua tabela de hash devido ao uso de arraylists. Uma alternativa ao encadeamento separado para evitar colisões é o *endereçamento aberto*.

Uma função de endereçamento aberto procura uma célula vazia na tabela de hash para nela colocar um item. Se a primeira célula encontrada pela função está em uso, tenta-se a próxima célula vazia, e assim por diante até que uma célula vazia seja, eventualmente, encontrada. Estudaremos duas estratégias diferentes para endereçamento aberto nesta seção: sondagem linear e sondagem quadrática.

A sondagem linear usa uma função linear para determinar a célula do vetor que será verificada para uma inserção. Isto significa que células serão verificadas sequencialmente até que uma célula vazia seja encontrada. O problema com a sondagem linear é que os dados tenderão a se agrupar em células adjacentes no vetor, tornando sucessivas sondagens por células vazias mais demorado e menos eficiente.

Quando usar a sondagem linear, o item será armazenado na próxima célula disponível na tabela, assumindo que a tabela ainda não esteja cheia.

Isso é implementado através de uma pesquisa sequencial por uma célula vazia a partir do ponto de colisão. Se o final físico da tabela for encontrado durante a pesquisa sequencial, a pesquisa deverá recomeçar no início da tabela e continuar a partir dai.

Se uma célula vazia não é encontrada antes de se retornar ao ponto de colisão, a tabela estará cheia.

[0]	72	[0]	72
[1]		[1]	15
[2]	18	[2]	18
[3]	43	[3]	43
[4]	36	[4]	36
[5]		[5]	10
[6]	6	[6]	6
[7]		[7]	5

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

[0]	49	[0]	49
[1]	58	[1]	58
[2]	69	[2]	69
[3]		[3]	
[4]		[4]	
[5]		[5]	
[6]		[6]	
[7]		[7]	
[8]	18	[8]	18
[9]	89	[9]	89

Um problema com o método de sondagem linear é que possível que blocos de dados se formem quando as colisões são resolvidas. Isso é chamado de agrupamento primário (**primary clustering**).

Uma característica ruim da sondagem linear é o fato de que, conforme a tabela vai sendo preenchida, blocos de células consecutivas se formam e o tempo exigido para a pesquisa aumenta proporcionalmente ao tamanho do bloco.

Além disso, quando tentamos inserir um item em uma posição da tabela que já esteja ocupada, esse item será finalmente incluído no fim do agrupamento -- portanto aumentando seu tamanho. Isto, por si só, não é inherentemente uma coisa ruim. Afinal, quando usamos a abordagem de bucket hash, cada inserção aumentava em uma unidade o tamanho de alguma das listas ligadas.

No entanto, sempre que uma inserção é feita entre dois agrupamentos que estão separados por uma posição vazia, os dois agrupamentos se tornarão um único, portanto potencialmente aumentando o comprimento do agrupamento por uma quantidade bem maior do que um -- isso é uma coisa ruim!

Isso significa que qualquer chave que seja espalhada no agrupamento exigirá várias tentativas para resolver a colisão.

Por exemplo, insira as chaves 89, 18, 49, 58 e 69 numa tabela de hash que mantenham 10 itens usando o método da divisão:

A **sondagem quadrática** elimina o problema de agrupamento. Uma função quadrática é usada para determinar que célula será verificada. Um exemplo de tal função é

$$\text{hash}(\text{chave}) + \text{collisionNumber}^2$$

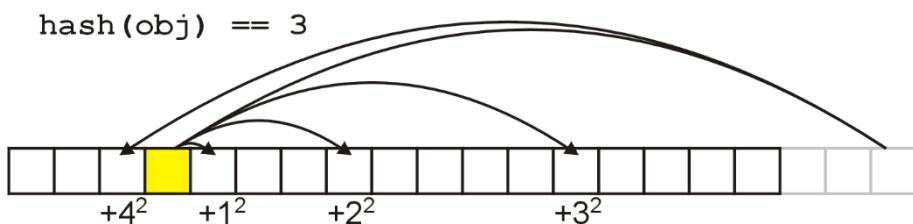
onde collisionNumber é o número de colisões que ocorreram durante a sondagem atual.
Uma propriedade interessante da sondagem quadrática é que ela garante que uma célula vazia será encontrada se a tabela de hash está menos do que metade vazia.

Com a sondagem quadrática, ao invés de sempre buscar uma célula em seguida, movemos i^2 células a partir do ponto de colisão, onde i é o número de tentativas de resolver a colisão.

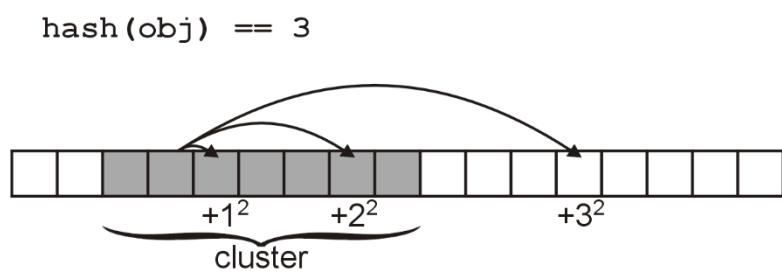
- Suponha que um elemento deveria ser colocado na célula h :
 - se a célula h está ocupada, então verifique a seguinte sequência de células:

$$h + 1^2, h + 2^2, h + 3^2, h + 4^2, h + 5^2, \dots$$

$$h + 1, h + 4, h + 9, h + 16, h + 25, \dots$$
- Por exemplo, para uma tabela com 17 posições (17 é primo!):



- Se um das sondagens $h + i^2$ cai em um agrupamento, isso não implica que a próxima sondagem também cairá:



89 % 10 = 9	[0]	49
18 % 10 = 8	[1]	
49 % 10 = 9 – 1 attempt needed – 1 ² = 1 spot	[2]	
58 % 10 = 8 – 3 attempts – 3 ² = 9 spots	[3]	69
69 % 10 = 9 – 2 attempts – 2 ² = 4 spots	[4]	
	[5]	
	[6]	
	[7]	58
	[8]	18
	[9]	89

Hashing Duplo

Esta estratégia simples para resolução das colisões é exatamente o que seu nome diz — se uma colisão ocorre, a função de hash é aplicada uma segunda vez e então sonda-se as células na sequência de distâncias $\text{hash}(\text{item})$, $2\text{hash}(\text{item})$, $4\text{hash}(\text{item})$, etc. até que uma célula vazia seja encontrada.

Para essa técnica de sondagem funcionar corretamente, algumas poucas condições devem ser garantidas. Primeiramente, a função de hash escolhida nunca deve resultar em zero, o que levaria a resultados desastrosos (visto que multiplicar por zero resulta em zero).

Em segundo lugar, o tamanho da tabela deve ser um número primo. Se o tamanho não é primo, então nem todas as células do vetor serão sondadas, novamente levando a resultados caóticos.

Hashing duplo é uma estratégia interessante para resolução de colisões, mas tem sido demonstrado na prática que a sondagem quadrática resulta em melhor desempenho.

Uma função popular de segundo hash é: $\text{Hash}_2(\text{chave}) = R - (\text{chave \% } R)$ onde R é um número primo menor que o tamanho da tabela.

Table Size = 10 elements	[0]	49
$\text{Hash}_1(\text{key}) = \text{key \% } 10$	[1]	
$\text{Hash}_2(\text{key}) = 7 - (\text{key \% } 7)$	[2]	
Insert keys: 89, 18, 49, 58, 69	[3]	69
	[4]	
$\text{Hash}(89) = 89 \% 10 = 9$	[5]	
$\text{Hash}(18) = 18 \% 10 = 8$	[6]	
$\text{Hash}(49) = 49 \% 10 = 9$ a collision !	[7]	58
$= 7 - (49 \% 7)$	[8]	18
$= 7$ positions from [9]	[9]	89
$\text{Hash}(58) = 58 \% 10 = 8$		
$= 7 - (58 \% 7)$		
$= 5$ positions from [8]		
$\text{Hash}(69) = 69 \% 10 = 9$		
$= 7 - (69 \% 7)$		
$= 1$ position from [9]		

Hashing com Rehashing

Uma vez que a tabela fique muito cheia, o tempo de execução das operações começará a ficar muito longo e poderá haver falhas. Para resolver esse problema, uma tabela com ao menos o dobro do tamanho da original deverá ser criada e os elementos deverão ser transferidos para a nova tabela.

O tamanho da nova tabela de hash:

- deve ser também um número primo
- deverá ser usado para calcular o novo local de inserção dos itens já incluídos na tabela original (dai vem o nome **rehashing**)

Essa é uma operação muito custosa! $O(N)$ já que haverá N itens para reespalhar e o tamanho da tabela é aproximadamente $2N$. Essa operação será aceitável desde que não ocorra frequentemente.

Quando devemos aplicar rehashing? Algumas possíveis respostas:

- quando a tabela ficar cheia pela metade
- Sempre que uma inserção falhar
- Sempre que um fator de carga específico seja alcançado, onde o fator de carga é a razão entre o número de elementos armazenados e o tamanho da tabela

Informações adicionais:

http://www.cse.yorku.ca/course_archive/2004-05/W/2011N/Notes/hash_tables_2.pdf

Animação da Tabela de Hash (com applets, poderá não funcionar):

<http://people.cs.pitt.edu/~kirk/cs1501/animations/Hashing.html>

5. Tabela de Hash Genérica

Interface IRegistro

Usada como protótipo de classes cujos dados serão lidos de arquivos e armazenados em estruturas de dados, como tabelas de Hash, Listas Ligadas, Pilhas, Filas e Árvores.

```
using System;
using System.IO;

public interface IRegistro<Tipo> where Tipo : IComparable<Tipo>
{
    Tipo LerRegistro(StreamReader arquivo);
    void EscreverRegistro(StreamWriter arquivo);
    int CompareTo(Tipo outro); // <0, ==0, >0
    string Chave { get; }
}
```

Exemplo de uma classe modelada a partir dessa interface: a classe de cidades de um mapa:

```
using System;
using System.IO;

public class Cidade : IRegistro<Cidade>, IComparable<Cidade>
{
    // mapeamento da linha de dados do arquivo texto

    const int tamanhoNome = 15,
              tamanhoX = 7,
              tamanhoY = 7,
```

```
    inicioNome = 0,
    inicioX = inicioNome + tamanhoNome,
    inicioY = inicioX + tamanhoX;

// atributos que formam uma linha do arquivo de cidades
string nome;
double x, y;

public Cidade() { } // construtor default

public Cidade LerRegistro(StreamReader arquivo)
{
    if (arquivo != null) // está aberto
    {
        string linha = arquivo.ReadLine(); // lê uma linha
        nome = linha.Substring(inicioNome, tamanhoNome);
        x = double.Parse(linha.Substring(inicioX, tamanhoX));
        y = double.Parse(linha.Substring(inicioY));
        return this;
    }
    return default(Cidade); // para arquivo não aberto
}

public void EscreverRegistro(StreamWriter arquivo)
{
    if (arquivo != null)
        arquivo.WriteLine($"{nome}{x:0.00000}{y:0.00000}");
}

public int CompareTo(Cidade outra) // <0, ==0, >0
{
    return this.nome.CompareTo(outra.nome);
}

public string Chave => this.nome;
}
```

Interface genérica para Tabela de Hash

Informará os métodos e propriedades que deverão ser implementados por qualquer classe de Tabela de Hash. Nesse momento, ainda não se sabe quais estratégias de hash serão usadas na implementação da tabela de hash. Ou seja, não se sabe **como** será feita a implementação, mas já se saberá **o que** deverá ser codificado.

```
int Hash(string chave);
void Inserir(Tipo item);
bool Remover(Tipo item);
bool Existe(Tipo item, out int onde);
List<string> Conteudo();
}
```

Implementação da Classe BucketHash seguindo a interface ITabelaDeHash

```
using System;
using System.Collections;
using System.Collections.Generic;

class BucketHash<Tipo> : ITabelaDeHash<Tipo>
    where Tipo : IRegistro<Tipo>, IComparable<Tipo>
{
    private const int SIZE = 131;
    ArrayList[] dados;

    public BucketHash()
    {
        dados = new ArrayList[SIZE];
        for (int i = 0; i < SIZE; i++)
            dados[i] = new ArrayList(1);
    }

    public int Hash(string chave)
    {
        long tot = 0;

        for (int i = 0; i < chave.Length; i++)
            tot += 37 * tot + (char)chave[i];

        tot = tot % dados.Length;
        if (tot < 0)
            tot += dados.Length;

        return (int)tot;
    }

    public void Inserir(Tipo item)
    {
        int valorDeHash = Hash(item.Chave);
        if (!dados[valorDeHash].Contains(item)) // Contains procura o item e
        retorna True ou False
            dados[valorDeHash].Add(item);
    }
}
```

```
public bool Remover(Tipo item)
{
    int onde = 0;
    if (!Existe(item, out onde))
        return false;
    dados[onde].Remove(item);
    return true;
}

public bool Existe(Tipo item, out int onde)
{
    onde = Hash(item.Chave);
    return dados[onde].Contains(item);
}

public List<string> Conteudo()
{
    List<string> saida = new List<string>();
    for (int i = 0; i < dados.Length; i++)
        if (dados[i].Count > 0)
    {
        string linha = $"{i,5} : ";
        foreach (Tipo item in dados[i])
            linha += " | " + item.Chave;
        saida.Add(linha);
    }
    return saida;
}

}
```

Classe de Hash com Sondagem Linear implementando a interface IHashTable

```
using System;
using System.Collections.Generic;

public class HashLinear<Tipo> : ITabelaDeHash<Tipo>
    where Tipo : IRegistro<Tipo>, IComparable<Tipo>
{
    public List<string> Conteudo()
    {
        throw new NotImplementedException();
    }

    public int Hash(string chave)
    {
        throw new NotImplementedException();
    }
}
```

```
bool ITablaDeHash<Tipo>.Existe(Tipo item, out int onde)
{
    throw new NotImplementedException();
}

void ITablaDeHash<Tipo>.Inserir(Tipo item)
{
    throw new NotImplementedException();
}

bool ITablaDeHash<Tipo>.Remover(Tipo item)
{
    throw new NotImplementedException();
}
```

Classe de Hash com Sondagem Quadrática implementando a interface IHashTable

```
using System;
using System.Collections.Generic;

public class HashQuadratico<Tipo> : ITablaDeHash<Tipo>
    where Tipo : IComparable<Tipo>, IRegistro<Tipo>
{
    public List<string> Conteudo()
    {
        throw new NotImplementedException();
    }

    public bool Existe(Tipo item, out int onde)
    {
        throw new NotImplementedException();
    }

    public int Hash(string chave)
    {
        throw new NotImplementedException();
    }

    public void Inserir(Tipo item)
    {
        throw new NotImplementedException();
    }

    public bool Remover(Tipo item)
    {
        throw new NotImplementedException();
    }
}
```

```
}
```

Classe de Hash Duplo implementando a interface IHashTable

```
using System;
using System.Collections.Generic;

public class HashDuplo<Tipo> : ITabelaDeHash<Tipo>
    where Tipo : IRegistro<Tipo>, IComparable<Tipo>
{
    public List<string> Conteudo()
    {
        throw new NotImplementedException();
    }

    public bool Existe(Tipo item, out int onde)
    {
        throw new NotImplementedException();
    }

    public int Hash(string chave)
    {
        throw new NotImplementedException();
    }

    public void Inserir(Tipo item)
    {
        throw new NotImplementedException();
    }

    public bool Remover(Tipo item)
    {
        throw new NotImplementedException();
    }
}
```

Trecho de Aplicação Windows Forms usando a Interface para declarar uma tabela de Hash

O trecho de código abaixo, ao invés de declarar 4 variáveis diferentes para as estratégias de Tabela de Hash, uma de cada classe específica (BucketHash, HashLinear, HashQuadratico, HashDuplo), apenas declara uma única variável, tabelaDeHash, cujo tipo é a interface ITabelaDeHash.

Como todas as classes citadas acima implementam (aderem a) a interface ITabelaDeHash, elas são compatíveis com essa interface e, portanto, a variável tabelaDeHash pode armazenar uma instância de qualquer das quatro classes, e o programa pode usar apenas os métodos previstos na interface ITabelaDeHash para tratar os dados que precisar armazenar. Dessa forma, se unifica o tratamento de uma tabela de Hash independentemente do tipo específico de tabela de Hash que for usada.

Nesse programa, o usuário poderá escolher qual técnica de Hash deseja, e o programa usará sempre a variável tabelaDeHash, sem se importar com qual classe de Hash ela armazena:

```
public partial class FrmCaminhos : Form
{
...
    ITablaDeHash<Cidade> tabelaDeHash;
...
    private void btnAbrirArquivo_Click(object sender, EventArgs e)
    {
        if (dlgAbrir.ShowDialog() == DialogResult.OK)
        {
            // verificamos qual a técnica de Hash escolhida
            // pelo usuário e criamos uma tabela de hash de
            // acordo com essa escolha
            if (rbBucketHash.Checked)
                tabelaDeHash = new BucketHash<Cidade>();
            else
                if (rbSondagemLinear.Checked)
                    tabelaDeHash = new HashLinear<Cidade>();
                else
                    if (rbSondagemQuadratica.Checked)
                        tabelaDeHash = new HashQuadratico<Cidade>();
                    else
                        tabelaDeHash = new HashDuplo<Cidade>();

            // abrimos o arquivo escolhido
            var asCidades = new StreamReader(dlgAbrir.FileName);
            // ler registros do arquivo aberto
            while (!asCidades.EndOfStream)
            {
                // instanciar um objeto cidade
                // lê-lo do arquivo para preencher seus atributos
                // armazenar esse objeto na tabela de Hash
                // de acordo com a técnica de hash escolhida
                // pelo usuário
                tabelaDeHash.Inserir(cidadeLida);
            }
            // Desenhar os nomes das cidades no mapa de Marte
            asCidades.Close(); // deixar arquivo fechado
        }
    }
...
}
```

6. Objeto Hashtable do C#

Terminamos de examinar implementações de tabelas de hash programadas por nós. No entanto, para a maioria das aplicações que usem C#, é melhor usar a classe Hashtable que já vem como parte da biblioteca do .NET Framework.

A classe Hashtable é um tipo especial do objeto Dictionary, armazenando pares chave-valor, onde os valores são armazenados baseando-se no código de hash derivado da chave. Você pode especificar uma função de hash ou usar aquela que já vem disponível (nós a discutiremos depois) para o tipo de dado da chave. A classe Hashtable é muito eficiente e deve ser usada no lugar de implementações desenvolvidas pelo programador da aplicação sempre que possível.

A estratégia que a classe usa para evitar colisões é o conceito de bucket.

Um bucket é um agrupamento virtual de objetos que tem o mesmo Código de hash, muito parecido como quando usamos um ArrayList para tratar colisões quando discutimos encadeamento separado. Se duas chaves têm o mesmo Código de hash, elas devem ser colocadas no mesmo bucket. Por outro lado, cada chave com um único Código de hash é colocada em seu próprio bucket.

O número de buckets usados em um objeto Hashtable é chamado de *fator de carga*.

O fator de carga é a razão entre a quantidade de elementos e o número de buckets. Inicialmente, o fator é ajustado como 1.0. Quando o fato real se encontra com o fator inicial, o fator de carga é incrementado para o menor número primo que seja duas vezes o número atual de buckets. O fator de carga é importante porque quanto menor o fator de carga, melhor desempenho terá o objeto Hashtable.

Instanciando e adicionando dados a um objeto Hashtable

A classe Hashtable é parte do namespace System.Collections, de forma que, para usá-la, você deve importar System.Collections no início de seu programa. Um objeto Hashtable pode ser instanciado de 3 formas (em verdade há muitas mais, incluindo diferentes tipos de construtores de cópia, mas nos restringiremos aqui aos três construtores mais comuns). Você pode instanciar a tabela de hash com uma capacidade inicial ou usar a capacidade default. Você também pode especificar a capacidade inicial e o fator de carga. O Código a seguir demonstra como usar esses três construtores:

```
Hashtable simbolos = new Hashtable();
Hashtable simbolos = new Hashtable(50);
Hashtable simbolos = new Hashtable(25, 3.0);
```

A primeira linha cria a tabela de hash com a capacidade default e o fator de carga default. A segunda linha cria uma tabela de hash com capacidade de 50 elementos e o fator de carga default. A terceira linha cria uma tabela de hash com uma capacidade inicial de 25 elementos e fator de carga 3.0.

Usa-se o método Add para incluir pares Chave-Valor em uma tabela de hash. Este método tem dois argumentos: a chave e o valor associado à essa chave. A chave é adicionada à tabela de hash após se calcular seu valor de hash. Eis um exemplo de código:

```
Hashtable simbolos = new Hashtable(25);
simbolos.Add("salario", 8000);
simbolos.Add("nome", "David Durante");
simbolos.Add("idade", 45);
simbolos.Add("depto", "Tecnologia da Informação");
```

Você também pode adicionar elementos a uma tabela de hash usando uma propriedade indexer, que estudamos no 1º semestre. Para fazer isso, você codifica o comando de atribuição que atribui um valor para a chave especificada como o índice (de forma semelhante a um índice de vetor). Se a chave ainda não existe, um novo elemento de hash é incluído na tabela; se a chave já existe, o valor existente é sobreescrito pelo novo valor. Seguem alguns exemplos:

```
simbolos["sexo"] = "M";
simbolos["idade"] = 44;
```

A primeira linha mostra como criar um novo par chave-valor usando o indexer, enquanto a segunda linha demonstra que você pode **sobrescrever** o valor atualmente associado a uma chave já existente.

Recuperando as Chaves e os Valores separadamente de uma Tabela de Hash

A classe Hashtable tem duas propriedades muito úteis para recuperar as chaves e os valores separadamente de uma tabela de hash: **Keys** e **Values**. Essas propriedades criam um objeto Enumerator que permite usar um loop For Each ou alguma outra técnica com índices para examinar as chaves e os valores.

O programa a seguir demonstra como essas propriedades funcionam:

```
using System;
using static System.Console;
using System.Collections;
namespace apHashTable01
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable simbolos = new Hashtable(25);
            simbolos.Add("salario", 8000);
            simbolos.Add("nome", "David Durante");
            simbolos.Add("idade", 45);
            simbolos.Add("depto", "Tecnologia da Informação");
            simbolos["sexo"] = "M";
            WriteLine("As chaves são: ");
            foreach (Object chave in simbolos.Keys)
                WriteLine(chave);
            WriteLine("\nAs chaves e seus valores são: ");
            foreach (DictionaryEntry par in simbolos)
                WriteLine($"{par.Key} = {par.Value}");
            Object valor = simbolos["nome"];
            WriteLine($"O valor da variável nome é {valor.ToString()}");
        }
    }
}
```

Recuperando um Valor com base na Chave

A recuperação de um valor através de sua chave associada pode ser feita usando um indexer, que atua exatamente como um índice de um vetor. Uma chave é passada como valor do índice e o valor associado a essa chave é retornado, a menos que a chave não exista, caso que se retorna **null**.

O próximo trecho de código demonstra como essa técnica funciona:

```
Object valor = simbolos["nome"];
WriteLine($"O valor da variável nome é {valor.ToString()}");
```

O valor retornado é “David Durante”. Podemos usar um indexer em conjunto com a propriedade Keys para recuperar todos os dados armazenados em uma hashtable. Colocamos esse código no final do programa acima:

```
WriteLine("\nConteúdo da tabela de hash:\n");
foreach (Object chave in simbolos.Keys)
    WriteLine(chave.ToString() + " : " +
        simbolos[chave].ToString());
```

Vemos a saída do programa na figura ao lado.

```
Console de Depuração do Microsoft Visual Studio
nome
depto
idade
sexo
salario

As chaves e seus valores são:
nome = David Durante
depto = Tecnologia da Informação
idade = 45
sexo = M
salario = 8000
O valor da variável nome é David Durante

Conteúdo da tabela de hash:

nome: David Durante
depto: Tecnologia da Informação
idade: 45
sexo: M
salario: 8000
```

Métodos Utilitário da classe Hashtable

Há vários métodos na classe Hashtable que lhe auxiliam a ser mais produtivo com objetos dessa classe. In this section, Examinaremos vários deles, incluindo métodos para determinar o número de elementos em uma hash table, limpar seu conteúdo, determinar se uma determina chave (e valor) está contida na tabela, remover elementos dessa tabela e copiar os elementos da tabela para um vetor.

O número de elementos em uma hash table é armazenado na propriedade Count, que retorna um inteiro:

```
int numElementos = simbolos.Count;
```

Podemos imediatamente remover todos os elevemtnos de uma hash ao usar o método Clear():

```
simbolos.Clear();
```

Para remover um único elemento de uma hash table, você pode usar o método Remove(). Este método tem um único argumento, uma chave, e remove tanto a chave especificada quanto seu valor associado. Por exemplo:

```
simbolos.Remove("sexo");
foreach (Object key in simbolos.Keys)
    WriteLine($"{key.ToString()} : {simbolos[key].ToString()}");
```

Antes que você remova um elemento de uma hash table, você pode desejar verificar se tanto a chave ou o valor estão na tabela. Esta informação pode ser determinada com os métodos ContainsKey() e ContainsValue(). O fragment de código abaixo demonstra como usar o método ContainsKey():

```
Write("Digite a chave a remover: ");
string umaChave = ReadLine();
if (simbolos.ContainsKey(umaChave))
    simbolos.Remove(umaChave);
```

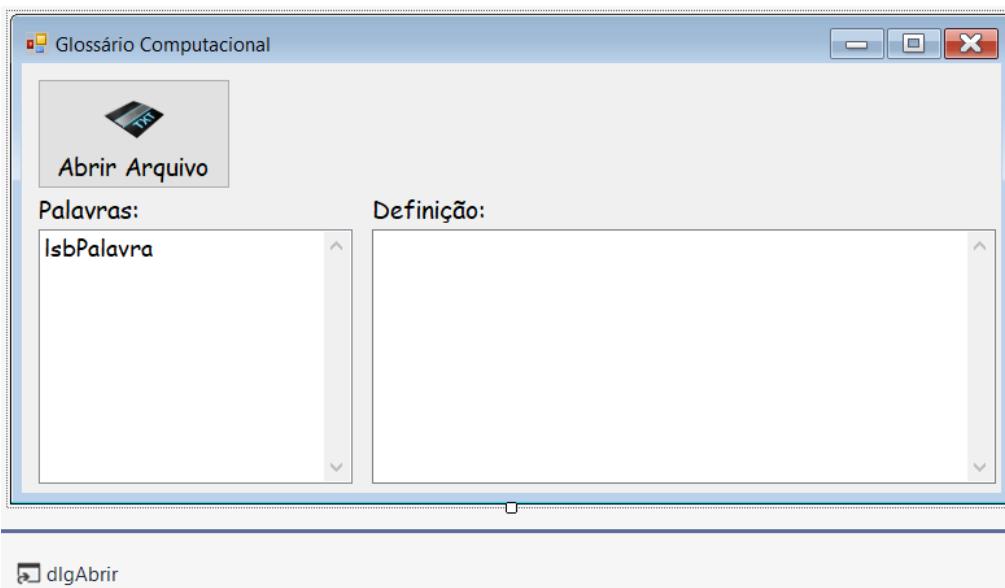
O uso deste método assegura que o par Chave-Valor a ser removido existe na hash table. O método ContainsValue() funciona de maneira similar com valores, ao invés de chaves.

Uma Aplicação com HashTable: Glossário de Termos Computacionais

Um uso comum de uma hash table é construir um glossário, ou dicionário, de termos. Nesta seção, demonstraremos uma maneira de usar uma hash table para esse uso — um glossário de termos computacionais.

Este programa funciona primeiramente lendo um conjunto de termos e definições a partir de um arquivo texto. Este processo é codificado na subrotina ConstruirGlossario. A estrutura do arquivo texto é: *palavra,definição*, com a vírgula no papel de delimitador entre uma palavra e sua definição. Cada palavra neste glossário é uma palavra única, mas ele poderia facilmente funcionar com frases. É por esse motivo que uma vírgula é usada como delimitador, ao invés de um espaço. Esta estrutura também nos permite usar a palavra como sendo a chave, que é a maneira correta de construir uma hash table. Outra subrotina, ExibirPalavras, exibe as palavras em um list box para que o usuário possa selecioná-la e, assim, obter uma definição. Já que as palavras são as chaves, podemos usar o método Keys para retornar apenas as palavras da hash table. O usuário poderá, assim, ver quais palavras possuem definições.

Para recuperar uma definição, o usuário simplesmente clicará em uma palavra do list box. A definição será recuperada usando o método Item e será exibida no text box. Abaixo temos um modelo do formulário:



O código do formulário e seus eventos vem abaixo:

```
using System;
using System.Collections;
using System.IO;
using System.Windows.Forms;

namespace apGlossario
{
    public partial class FrmGlossario : Form
    {
        private Hashtable glossario = new Hashtable();
        public FrmGlossario()
        {
            InitializeComponent();
        }
    }
}
```

```
}

private void ConstruirGlossario(Hashtable tabela, string nomeArquivo)
{
    StreamReader arq = new StreamReader(nomeArquivo,
                                         System.Text.Encoding.UTF7);
    string[] cadeiasLidas;

    char[] delimitador = new char[] { ',' };
    while (!arq.EndOfStream)
    {
        string linha = arq.ReadLine();
        cadeiasLidas = linha.Split(delimitador);
        tabela.Add(cadeiasLidas[0], cadeiasLidas[1]);
    }
    arq.Close();
}

private void btnAbrirArquivo_Click(object sender, EventArgs e)
{
    if (dlgAbrir.ShowDialog() == DialogResult.OK)
    {
        ConstruirGlossario(glossario, dlgAbrir.FileName);
        lsbPalavra.Enabled = true;
        ExibirPalavras(glossario, lsbPalavra);
    }
}

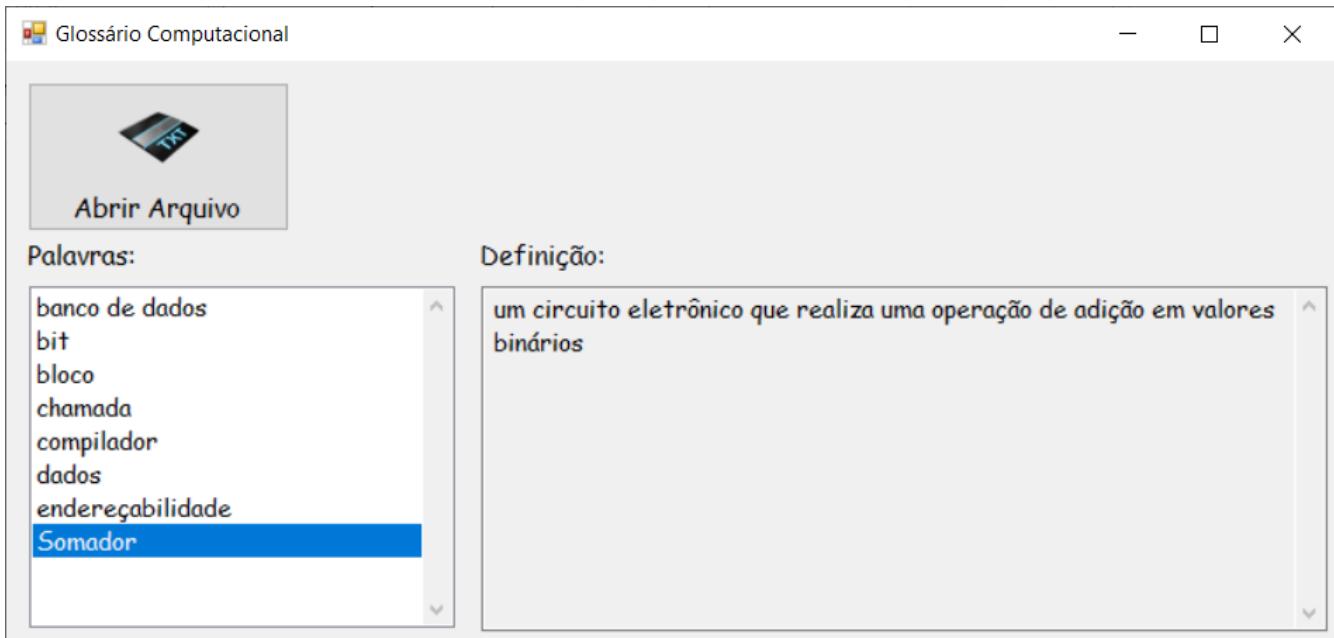
private void ExibirPalavras(Hashtable tabela, ListBox lista)
{
    Object[] palavras = new Object[1000];
    tabela.Keys.CopyTo(palavras, 0);
    for (int i = 0; i < palavras.Length; i++)
        if (palavras[i] != null)
            lista.Items.Add(palavras[i]);
}

private void lsbPalavra_Click(object sender, EventArgs e)
{
    Object palavra = lsbPalavra.SelectedItem;
    txtDefinicao.Text = glossario[palavra].ToString();
}
}
```

O arquivo texto será semelhante ao texto abaixo:

Somador,um circuito eletrônico que realiza uma operação de adição em valores binários endereçabilidade, o número de bits armazenados em cada localização endereçável da memória bit,abreviação de binary digit
bloco,um grupo lógico de zero ou mais comandos de programa
chamada,o ponto a partir do qual o computador começa a seguir as instruções em um subprograma
compilador,um programa que traduz um programa de alto nível em código de máquina
dados,informações em um formato que um computador pode utilizar
banco de dados,um conjunto estruturado de dados

Exemplo de execução:



Uma tabela de hash é uma estrutura muito eficiente para armazenar pares Chave-Valor de dados. A implementação de uma tabela de hash é bastante fácil, com a parte complicada sendo a escolha de uma estratégia para tratar as colisões. Aqui discutimos várias técnicas para tratá-las.

Para a maioria das aplicações C#, não há razão para programar uma hash table, já que a classe Hashtable da biblioteca do .NET Framework funciona tão bem.

Você pode especificar sua propria função de hash ou deixar que a classe calcule os valores de hash segundo suas funções já previamente embutidas.

5. Filas

1. Conceitos fundamentais

Uma fila é uma estrutura de dados que representa situações onde o fluxo de tratamento de dados é da seguinte forma:

- o primeiro elemento que entra na fila é o primeiro a ser atendido pelo processo.
- todo novo elemento que aparece sempre entra ao final da fila.
- o último elemento a entrar na fila será o último a ser atendido pelo processo.

Essas regras determinam uma disciplina de acesso, em que o último que entra é o último a ser atendido; o nome mais comum para essa disciplina é **FIFO**, ou First In, First Out (o primeiro que entra é o primeiro que sai).

Geralmente, uma fila é manipulada de acordo com a disciplina acima descrita. No entanto, pode-se estabelecer prioridades para cada elemento, de modo que um elemento possa entrar no meio da fila, antes de um de prioridade menor. Quando a prioridade é a mesma, pode-se ter um critério de desempate, ou, então, o que é mais usual, assume-se o critério de ordem cronológica de chegada. Quem chegou depois fica após quem chegou anteriormente.

Assim, uma fila pode ter varias subfilas, de acordo com os níveis de prioridades estabelecidas. No entanto, para se retirar um elemento, este, quase sempre, será o primeiro da fila.

Como exemplos de filas no dia-a-dia, temos as filas de supermercados, caixas de bancos, filas em postos de saúde, dentre várias outras. Já no mundo da computação, existem diversas aplicações de filas, como a fila de impressão de um computador servidor de impressão, que enfileira os pedidos de impressão dos vários usuários do sistema, atribuindo o local de cada arquivo para impressão de acordo com a prioridade do usuário, e a própria fila de processos esperando para ser executados num computador com um único processador. Além disso, algoritmos de tratamento do fluxo de transporte de dados em redes de computadores também usam filas para armazenar pacotes que chegaram a um computador, transmitidos através da rede, mas que ainda não puderam ser processados.

2. Representação de Filas

Uma fila é um objeto abstrato composto por três elementos: a **área** de memória onde serão armazenados os dados e duas variáveis, que chamaremos de indicador de **início** e indicador de **fim** da fila, respectivamente.

Essas variáveis informam, a cada momento, a localização de onde se pode retirar elementos da fila (o início) e onde se pode incluir novos elementos na fila (o fim).

Assim sendo, primeiramente as únicas posições acessadas na fila serão a posição indicada pelo início e a posição indicada pelo fim, e acessos internos a posições da fila não acontecerão frequentemente. Uma representação de fila, portanto, precisa armazenar dados e aplicar sobre esses dados a disciplina de acesso dessa estrutura de dados, através dos métodos fundamentais que descrevemos a seguir:

Enfileirar(o)	→	Insere o objeto genérico o no final da fila Recebe : objeto genérico o Devolve : nada
Retirar()	→	retira da fila o objeto que está em seu início e retorna esse objeto Recebe : nenhuma Devolve : objeto genérico retirado do início da fila

Note que no método anterior, a fila não poderá estar vazia, para que se possa retirar um objeto de seu início. Portanto, caso a fila esteja vazia, será necessário disparar uma exceção, para que o programa que usa a fila trate-a de maneira coerente com os objetivos desse programa. Isso nos leva a pensar em alguns métodos adicionais, que listamos a seguir:

Tamanho()	→	devolve o número de objetos enfileirados na fila Recebe : nada Devolve : inteiro
EstaVazia()	→	devolve true se a fila estiver vazia, e false caso contrário Recebe : nada Devolve : valor lógico
oInício()	→	devolve o objeto no início da fila, sem retirá-lo Recebe : nada Devolve : objeto genérico
oFim()	→	devolve o objeto no final da fila, sem retirá-lo Recebe : nada Devolve : objeto genérico

Vamos definir uma interface genérica, que apresente os métodos básicos independentemente do tipo de armazenamento escolhido para implementação (se vetor ou lista ligada). Isso cria uma “camada de isolamento” entre a implementação da estrutura de dados e o programa de aplicação que utiliza a fila, de maneira que os detalhes da implementação internos ficam desconhecidos da aplicação, e apenas os nomes de métodos são conhecidos e acessíveis. A nossa interface não especificará que tipos de objetos podem ser armazenados na fila, de modo que qualquer objeto poderá ser nela armazenado. Abaixo temos a descrição da interface:

```
public interface IQueue<Tipo>
{
    void Enfileirar(Tipo elemento); // inclui objeto "elemento"
    Tipo Retirar(); // devolve objeto do início e o
                    // retira da fila
    Tipo OInício(); // devolve objeto do início
                    // sem retirá-lo da fila
    Tipo OFim(); // devolve objeto do fim
                    // sem retirá-lo da fila
    int Tamanho { get; } // devolve número de elementos da fila
    bool EstaVazia { get; } // informa se a fila está vazia ou não
}
```

Quando ocorrer de se buscar o elemento do início da fila e esta estiver vazia, deve-se disparar uma exceção, que cuidará dessa situação de erro. Essa exceção é definida pelo código abaixo:

```
public class FilaVaziaException : Exception
{
    // chama o método da classe ancestral (classe base) de Exception
    public FilaVaziaException (String erro) : base(erro)
    { }
}
```

Após termos definido a interface, que informa os métodos que realizam as operações de uma fila, devemos definir uma classe que implemente os métodos acima. Essa implementação pode ser feita de várias maneiras, e o importante aqui é notar que, qualquer que seja a maneira de implementar essas operações internamente, a interface com a aplicação continua sendo aquela definida acima. Isso permite que a aplicação funcione normalmente mesmo que a implementação da estrutura de dados seja modificada, isolando a aplicação de detalhes de implementação.

Assim, podemos concluir que uma estrutura de dados compatível com a definição de fila é um objeto que contenha a **área de armazenamento**, que pode ser um vetor, e o indicador do **Fim** da fila. Pode-se usar outras estruturas para a área de armazenamento, como uma lista ligada. Mas, para efeito de simplicidade, neste momento usaremos um vetor, como vemos na figura abaixo:

fim = 2

0	1	2	3	4	5	6
João	Maria	Fernando	vazio	vazio	vazio	Vazio

Figura 28 – representação de uma fila de pessoas

Um vetor possui certas características que facilitam a implementação, pois torna-a muito simples. No entanto, um vetor tem que possuir um número de elementos previamente definido quando é criado, de modo que quando usamos um vetor para implementar a fila, precisamos definir seu tamanho N. Assim, nossa fila poderá ser composta por um **vetor F de N posições** e a variável **fim** que informará o índice do elemento final da fila.

Dessa maneira, supomos que o início da fila sempre será a posição 0 do vetor. Quando a fila é criada, o final da mesma estaria na posição -1, de maneira que a fila estaria vazia. Quando incluíssemos um novo elemento na fila, ele seria colocado na posição seguinte ao índice de final da fila (**fim**), e este seria incrementado. Esse processo é visto na figura abaixo:

fim = -1	0	1	2	3	4	5	6
	vazio						

Enfileirar("João")

fim = 0

fim = 0	0	1	2	3	4	5	6
	João	vazio	vazio	vazio	vazio	Vazio	Vazio

Enfileirar("Maria")

fim = 1

fim = 1	0	1	2	3	4	5	6
	João	Maria	vazio	vazio	vazio	Vazio	Vazio

Enfileirar("Fernando")

fim = 2

fim = 2	0	1	2	3	4	5	6
	João	Maria	Fernando	vazio	vazio	Vazio	Vazio

Figura 11 – Inclusão de elementos numa fila implementada com vetor

No entanto, quando houver necessidade de retirar um elemento da fila, como isso será feito? Como, neste modelo em discussão, estamos supondo que o elemento 0 é sempre o início da fila, haveria a necessidade de deslocar os elementos seguintes ao primeiro, para a posição anterior a cada um deles. Isso é feito com um comando **for**, como podemos ver no algoritmo abaixo:

```
fim--; // decrementa fim, pois deslocará todos os elementos
for ( indice=0; indice<fim; indice++)
    F[indice] = F[indice+1];
```

Dependendo de quantos elementos houver na fila, esse processo será lento, pois teremos de deslocar todos os elementos existentes na fila. Na figura 12 vemos o deslocamento de elementos para retirada do primeiro da fila:

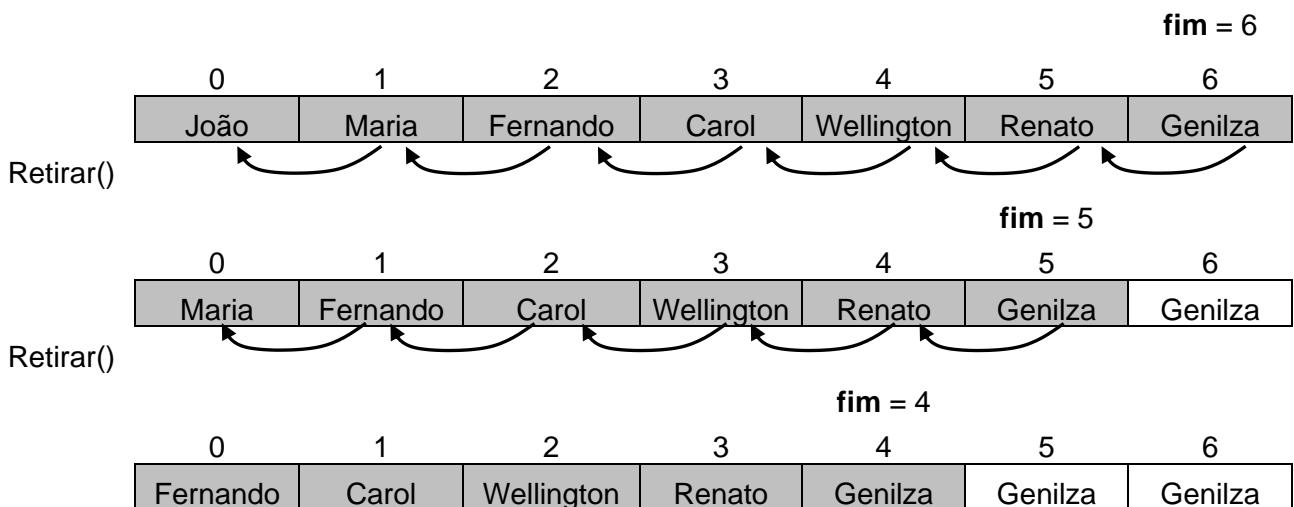


Figura 12 – retirada de elementos de uma fila, usando deslocamento

Observe que não é necessário “limpar” as posições finais do vetor, não mais usadas, pois o índice **fim** informa qual é a última posição válida, e nunca usaremos uma posição subsequente, a menos de quando desejamos incluir um novo elemento; nesse caso, o novo elemento por si só limpa a posição, ao ter seu conteúdo atribuído para ela.

Além disso, observamos que essa maneira de implementar nos obriga a deslocar os N elementos armazenados na fila, sempre que desejamos retirar o elemento inicial da mesma. Concluímos, portanto, que esse processo é um tanto quanto caro (lento), de ponto de vista computacional.

Para tornar o processo mais rápido, ao invés de fixar o início da fila como a posição 0 do vetor, uma alternativa é usarmos um **índice para o início** da fila, de maneira que simplesmente incrementamos esse índice quando retiramos o primeiro elemento da fila e desejamos indicar o elemento seguinte como sendo o novo primeiro elemento. Usamos também um índice **fim** para a primeira posição **livre** da fila, como vemos na figura 13:



Figura 13 – implementação de fila em vetor, com índices para Inicio e para Fim da fila

A figura 13 representa uma fila implementada com um vetor, em que o **início** está posicionado na posição 0 e a primeira posição livre sendo a 6, onde seria colocado qualquer novo elemento. Caso retiremos um elemento da fila, ele será retirado da posição 0, e a variável **Início** deverá indicar a posição seguinte, ou seja, a posição 1. Isso é o que ocorre na parte (a) da figura. Em seguida, retiramos mais um elemento (na parte (b) da figura) e o índice de **início** passa a apontar a posição 2 do vetor, que finalmente fica com a configuração exibida em (c).

Como o índice de Início, após uma retirada de elemento, é posicionado no elemento seguinte, não é necessário "limpar" o conteúdo de um elemento retirado da fila.

Quando a fila estiver vazia, **início** será igual a **fim**. Dessa maneira, a inclusão e a remoção de elementos ocorrem rapidamente, pois apenas um elemento é acessado e não existem deslocamentos.

No entanto, essa abordagem possui um problema adicional: como usamos um vetor, e como um vetor exige um tamanho fixo durante sua criação, temos o limite N de posições do vetor, de modo que a fila estará cheia após N inclusões. Problema semelhante ocorreria se tentássemos incluir e excluir o mesmo elemento N vezes: teríamos uma fila praticamente vazia, mas sem mais locais para realizar inclusões, pois **início** seria igual a **fim** e ambos iguais a N.

A maneira de evitar esse problema seria fazer com que início e fim “dessem a volta” no vetor quando necessário, de forma que tivéssemos um “vetor circular”, como podemos observar abaixo:

	início								fim
	0	1	2	3	4	5	6	7	
(a) Enfileirar("Lucas")				Carlos	Joana	Célia	Lucas		
	fim								início
	0	1	2	3	4	5	6	7	
(b) Enfileirar("Sônia")				Carlos	Joana	Célia	Lucas	Sônia	
	fim								início
	0	1	2	3	4	5	6	7	
(c) Enfileirar("Sérgio")	Sérgio				Joana	Célia	Lucas	Sônia	

Figura 14 – vetor circular na implementação de uma fila

Para garantir essa “circularização” do vetor, devemos proceder da seguinte maneira: no momento de incrementar **início** ou **fim**, esse incremento deve ser calculado como $(\text{inicio} + 1) \% N$ ou $(\text{fim} + 1) \% N$, respectivamente, onde N é o número de posições físicas do vetor e % é a operação de resto de divisão.

O único problema que nos resta é quando incluímos N elementos e não retiramos nenhum, de modo que **início** e **fim** ficarão idênticos e não teremos como distinguir uma fila vazia de uma fila cheia.

Uma maneira de detectar esse problema é impedir a inclusão de elementos quando já tivermos N – 1 elementos enfileirados, e disparar uma exceção de fila cheia, nesse caso.

Tendo essa discussão estabelecida, podemos passar à descrição dos algoritmos dos métodos e, posteriormente, à criação do objeto FilaVetor e seus métodos em linguagem C#.

```

Tamanho()
    retorno (N - inicio + fim) % N

EstaVazia()
    retorno (inicio == fim)

OInicio()
    se (EstaVazia()) então
        disparar uma FilaVaziaException
    retorno F[inicio]

OFim()

```

```

        se (EstaVazia()) então
            disparar uma FilaVaziaException
        se (fim == 0) então
            retorno F[N-1]
        senão
            retorno F[fim-1]

Enfileirar(o)
    se (Tamanho() == N -1 ) então
        disparar uma FilaCheiaException
    F[fim] ← o
    fim ← (fim + 1) % N

Retirar()
    se (EstaVazia())
        disparar uma FilaVaziaException
    o ← F[inicio]
    F[inicio] ← null // aqui estamos limpando mas não é necessário
    inicio ← (inicio + 1) % N
    retorno o

```

A seguir temos a classe, em C#, que implementa os métodos acima descritos.

```

public class FilaVetor<Tipo> : IQueue<Tipo>
{
    public const int MAXIMO = 500;    // tamanho default do vetor F
    int posicoes;                  // tamanho dado pela aplicação
    Tipo[] F; // vetor de objetos genéricos, com tamanho genérico,
               // usado como área de armazenamento
    int inicio = 0, // índice do início da fila
        fim = 0; // índice do fim da fila

    public FilaVetor() : this(MAXIMO) // construtor que utiliza o default MAXIMO
    {
        // chama o método construtor com parâmetro
        // utilizando o polimorfismo do construtor
    }

    public FilaVetor(int posic) // construtor polimórfico
    {
        posicoes = posic;          // armazena o tamanho físico do vetor
        F = new Tipo[posicoes];   // F é um vetor de Tipo; cria um
        // vetor F com o tamanho indicado

    // devolve o número de posições em uso
    public int Tamanho { get => (posicoes - inicio + fim) % posicoes; }

    // devolve true se fila está vazia, e false caso contrário
    public bool EstaVazia { get => (inicio == fim); }

    public Tipo OInicio()
    {
        if (EstaVazia)
            throw new FilaVaziaException("Esvaziamento (underflow) da fila");
        Tipo o = F[inicio];           // devolve o objeto do início da fila
        return o;                     // sem retirá-lo da fila
    }

    public Tipo OFim()
    {
        Tipo o;
        if (EstaVazia)
            throw new FilaVaziaException("Underflow da fila");
        if (fim == 0)
            o = F[posicoes-1];        // devolve o objeto do final da fila
    }
}

```

```

else                                // sem retirá-lo da fila
    o = F[fim-1];
return o;
}

public void Enfileirar(Tipo elemento)
{
    if (Tamanho == posicoes - 1)
        throw new FilaCheiaException("Fila cheia (overflow)");
    F[fim] = elemento;                // inclui elemento na primeira posição livre
    fim = (fim + 1) % posicoes; // calcula próxima posição livre
}

public Tipo Retirar()
{
    Tipo o;
    if (EstaVazia)
        throw new FilaVaziaException("Underflow da fila");
    o = F[inicio];                  // copia o elemento inicial da fila
    F[inicio] = default(Tipo);      // libera memória
    inicio = (inicio +1) % posicoes; // calcula novo inicio da fila
    return o;                      // devolve elemento inicial
}
}

```

Observe que, em C#, não se declara na assinatura de um método que este dispara uma exceção. Isso é feito apenas pelo comando **throw**.

3. Fila baseada em Lista Ligada

Da mesma maneira que acontece com pilhas, podemos usar listas ligadas para implementar filas. Nesse caso, as inclusões serão realizadas no final da fila e as remoções, em seu início, para não termos de ficar percorrendo a lista sempre que for necessário incluir ou remover algum elemento.

Essas alterações foram feitas para que possamos dispor de métodos que serão usados em FilaLista. Note que ela estenderá a funcionalidade da classe ListaSimples ao mesmo tempo em que implementa a interface Quere. Abaixo temos a implementação da interface Queue, baseada em lista ligada, usando a classe FilaLista:

```

class FilaLista<Tipo> : ListaSimples<Tipo>, IQueue<Tipo>
{
    where Tipo : IComparable<Tipo>

    public void Enfileirar(Tipo elemento) // inclui objeto "elemento"
    {
        NoLista<Tipo> novoNo = new NoLista<Tipo>(elemento, null);
        base.inserirAposFim(novoNo);
    }

    public Tipo Retirar()           // devolve objeto do início e o
    {                            // retira da fila
        if (!estaVazia())
        {
            Tipo elemento = base.Primeiro.info;
            base.removerNo(Primeiro, null);
            return elemento;
        }
        throw new FilaVaziaException("Fila vazia");
    }

    public Tipo OInicio()         // devolve objeto do início
    {                            // sem retirá-lo da fila
        if (estaVazia())
    }
}

```

```

        throw new FilaVaziaException("Fila vazia");
        Tipo o = base.Primeiro.Info; // acessa o 1º objeto genérico
        return o;
    }

public Tipo OFim()           // devolve objeto do fim
{                           // sem retirá-lo da fila
    if (estaVazia())
        throw new FilaVaziaException("Fila vazia");

    Tipo o = base.Ultimo.Info; // acessa o 1º objeto genérico
    return o;
}

// devolve número de elementos da fila
public int Tamanho { get => base.QuantosNos; }

public bool EstaVazia { get => base.EstaVazia; }

public List<Tipo> Listar()
{
    List<Tipo>[] itens = new List<Tipo>[this.Tamanho];
    int indice = 0;
    base.iniciarPercorsoSequencial();
    while (base.PodePercorrer())
    {
        itens[indice++] = base.Atual.Info.ToString();
    }
    return itens;
}
}

```

A mesma classe, implementada em Java, segue abaixo:

A figura abaixo ilustra a aparência dos objetos da fila:

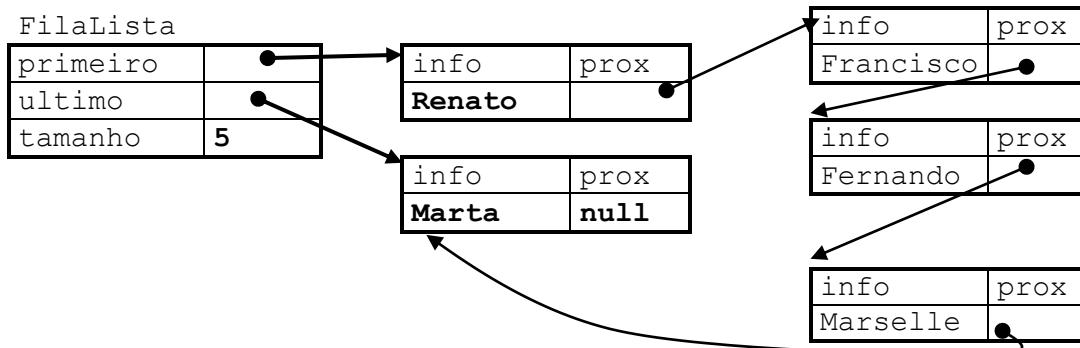


Figura 30 – fila implementada a partir de uma lista ligada

4. Exercícios

1. (2/1986) Durante um treinamento de naufrágio em um navio transatlântico foi estabelecida uma fila de embarque nos botes salva-vidas. As pessoas teriam seu lugar na fila de acordo com as seguintes prioridades:

1. mulheres até 15 anos
2. homens até 15 anos
3. mulheres de 16 a 35 anos
4. mulheres de 36 anos ou mais
5. homens de 16 ou mais

Cada pessoa que chegasse à fila ficaria sendo a última do seu grupo de prioridades, ou seja, seria colocada antes do primeiro do grupo seguinte.

Existem 10 botes, cada um com capacidade para 10 pessoas. Só se pode encher um bote de cada vez. Quando um é preenchido, outro é trazido, até que os 10 botes tenham partido, ou não haja mais pessoas para embarcar.

Faça uma aplicação Windows Forms em C# que simule este treinamento, em que a fila é montada a partir de um arquivo texto com os seguintes dados: Nome (30 posições), Idade (3 posições) e Gênero (1 posição). Os dados não estão ordenados por prioridade, de modo que você deve colocar cada pessoa na sua posição correta na fila, conforme o arquivo vai sendo lido.

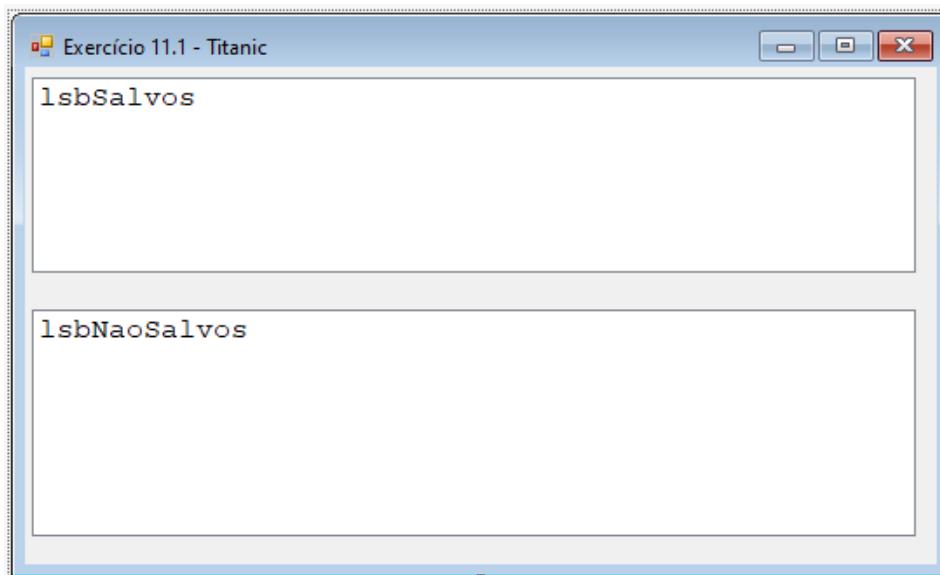
Após montar a fila, fazer o desembarque nos botes. Isto será representado pelo seguinte relatório:

Bote 1	Nome	Idade	Gênero
:	:	:	:
:	:	:	:
Bote 2	Nome	Idade	Gênero
:	:	:	:
:	:	:	:
...			
Bote 10	Nome	Idade	Gênero
:	:	:	:

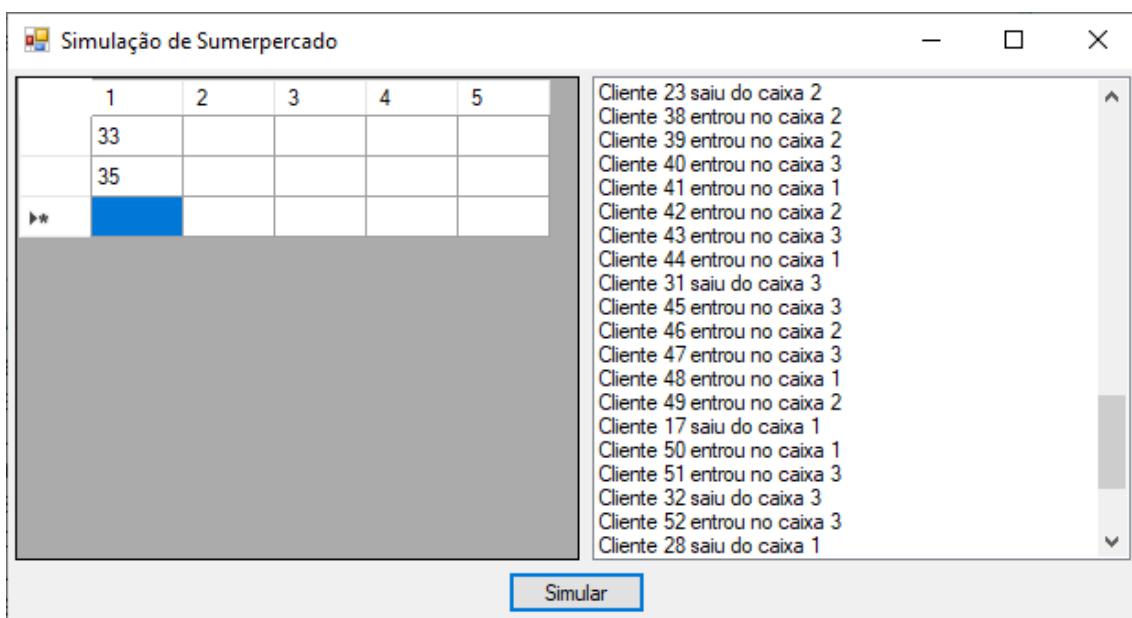
O relatório indicará os ocupantes de cada um dos botes usados.

Se houver pessoas que não consigam embarcar nos botes, imprima seus dados para o obituário naval. Ao final, entregue as listagens para Jones, o gato de bordo, e reze para a água não estar muito fria.

A lista dos passageiros salvos e a lista dos passageiros não-salvos podem ser apresentadas em Listboxes, como no modelo de formulário abaixo:



2. Faça uma aplicação Windows Forms em C# que simule o seguinte processo: num supermercado há 5 caixas. Os clientes se dirigem à fila do caixa que estiver mais vazia. Sempre aparece um novo cliente quando um número aleatório real sorteado é menor que 0.9, e este deve ser colocado na fila do caixa menos ocupado. Um cliente é atendido e liberado sempre de um caixa sorteado (número do caixa sendo sorteado entre 0 e 4, aleatoriamente), mas apenas se um número aleatório real sorteado é menor que 0.51. Cada novo cliente que aparece é identificado por um número sequencial, incrementando a cada novo cliente. O processo se inicia com todos os caixas vazios, e deve terminar quando não aparecer nenhum cliente após 5 rodadas consecutivas.



Sempre que um cliente entrar ou sair, deve-se exibir uma mensagem numa lista de mensagens, como o Listbox à direita na figura acima. A situação de cada um dos 5 caixas em cada momento da simulação deve ser exibida num DataGridView, como também vemos na figura anterior.

Não se deve declarar 5 filas individuais, mas sim um vetor de 5 filas, onde cada fila será armazenada. Esse vetor será percorrido sequencialmente para se descobrir qual das filas tem o menor tamanho.

Codifique na aplicação um método `ExibirFila()`, que recebe como um inteiro, que representa o número da fila (0 a 4). Esse método acessará o conteúdo dessa fila e percorrerá sequencialmente esse conteúdo e o apresentará na coluna do Datagridview que corresponde ao número da fila.

3. O Estacionamento do Cotuca contém uma única alameda que guarda até 20 carros. Os carros entram pela extremidade sul do estacionamento e saem pela extremidade norte. Se chegar um cliente para retirar um carro que não esteja estacionado logo na posição de saída, todos os carros ao norte do carro desejado serão deslocados para fora, o carro sairá do estacionamento e os outros carros voltarão à mesma ordem em que se encontravam inicialmente. Sempre que um carro deixa o estacionamento, todos os carros ao sul são deslocados para frente, de modo que, o tempo inteiro, todos os espaços vazios estarão na parte sul do estacionamento.

Escreva uma aplicação Windows Forms em C# que leia um grupo de linhas de um arquivo texto entrada. Cada linha contém um primeiro caracter 'C', de chegada, ou um 'P', de partida, além de uma placa de um carro que vem em seguida ao primeiro caracter. Presume-se que os carros chegarão e partirão na ordem especificada na entrada de dados. A aplicação deve exibir uma mensagem cada vez que um carro chegar ou partir. Quando um carro chegar, a mensagem deverá especificar se existe ou não vaga para o carro. Quando houver espaço disponível, outra mensagem deverá ser impressa. Quando um carro partir, a mensagem deverá incluir o número de vezes que o carro foi manobrado, tendo saído do estacionamento e voltado, incluindo a própria partida, mas não a chegada. Esse número será 0 se o carro for embora a partir da linha de espera.

4. Uma deque (double-ended queue), é uma fila em que os itens podem ser incluídos ou excluídos em qualquer das extremidades, mas nenhuma outra mudança pode ser efetuada no interior da fila. Tendo isso em mente, escreva métodos que permitam efetuar as operações abaixo e mantendo a disciplina de acesso descrita para uma fila:

- a. Incluir elemento no final da fila
- b. Incluir elemento no início da fila
- c. Excluir elemento no início da fila
- d. Excluir elemento no final da fila

6. Pilhas

1. Conceitos fundamentais

Pilhas são estruturas de dados usadas para armazenamento **temporário** de dados que devem ser recuperados na **ordem inversa** à que foram armazenados. O armazenamento e retirada de dados de uma pilha são realizadas de acordo com a disciplina de acesso **LIFO** (Last In First Out), que significa que o último elemento colocado será o primeiro a ser retirado. Esta disciplina de acesso é semelhante ao empilhamento de pratos: para se colocar um prato, coloca-se sobre a pilha. Para retirar, o que será retirado é o que foi colocado por último.

Pilhas são frequentemente usadas em compiladores para decodificar expressões aritméticas, calcular endereços de retorno de chamadas de procedimentos, armazenar variáveis locais de procedimentos e parâmetros, em sistemas operacionais e também na Máquina Virtual Java.

Outros usos são a resolução de algoritmos de pesquisas e apoio à decisão de melhor caminho dentre várias possibilidades.

Outro exemplo de uso de pilhas são os navegadores da Internet, que guardam os endereços mais recentemente visitados. Sempre que o usuário “entra” num site, seu endereço é guardado numa pilha (“empilhado”) e, quando o usuário resolve pressionar o botão [Voltar] ou [Back], retorna-se ao endereço que tinha sido empilhado por último (é “desempilhado”, e deixa de estar na pilha).

2. Representação de Pilhas

Uma pilha é um objeto abstrato composto por dois elementos: a área de memória onde serão armazenados os dados, e uma variável, que chamaremos de apontador de pilha (Stack Pointer).

Essa variável informa, a cada momento, a localização (endereço) do **topo da pilha**. O topo da pilha indica o local sobre o qual será empilhado o próximo elemento, e também indica o local de onde será retirado um elemento quando se desejar desempilhá-lo.

Assim sendo, a única posição acessada na pilha será a posição indicada pelo seu topo, e acessos a posições inferiores ao topo não acontecerão frequentemente. Uma representação de pilha seria:

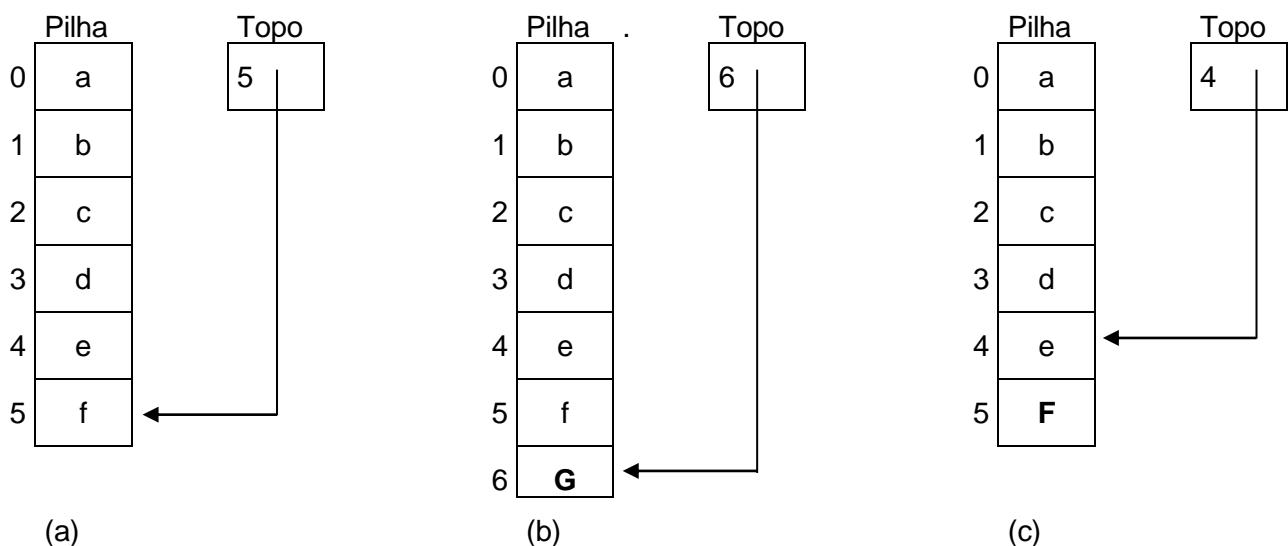


Figura 3 – representação de uma pilha

A figura acima representa uma pilha, em três ocasiões em seqüência, sendo que no momento inicial (a) estão empilhados 6 elementos. A variável TOPO contém o endereço da última posição usada da pilha, ou seja, o seu topo. Caso se coloque mais um elemento na pilha, o valor de TOPO deverá ser 6 (momento (b)).

Caso se retire um elemento da pilha, TOPO deverá cair para 4 (momento (c)).

Como se nota pela figura, não é necessário "limpar" o conteúdo de um elemento retirado da pilha, pois, como o conteúdo de TOPO diminui nesse processo, no momento de se colocar novo elemento na pilha, o TOPO "subirá", e o novo elemento será colocado na posição ocupada anteriormente, apagando o conteúdo não mais válido.

Portanto, uma pilha possui duas operações (métodos) principais: **empilhar()**, que permite armazenar informações e **desempilhar()** recuperar informações empilhadas.

Podemos definir esses métodos de uma maneira mais formal, facilitando sua posterior implementação como uma classe em Java:

- | | |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------|
| empilhar(o) → | Insere o objeto genérico o no topo da pilha
Recebe : objeto genérico o
Devolve : nada |
| desempilhar() → | retira da pilha o objeto que está em seu topo e retorna esse objeto
Recebe : nenhuma
Devolve : objeto genérico desempilhado |

Note que no método anterior, a pilha não poderá estar vazia, para que se possa retirar um objeto de seu topo. Portanto, caso a pilha esteja vazia, será necessário disparar uma exceção, para que o programa que usa a pilha trate-a de maneira coerente com os objetivos desse programa. Isso nos leva a pensar em alguns métodos adicionais, que listamos a seguir:

- | | |
|---------------|------------------------------------------------------------------------------------------------------------------------|
| tamanho() → | devolve o número de objetos na pilha
Recebe : nada
Devolve : inteiro |
| estaVazia() → | devolve true se a pilha estiver vazia, e false caso contrário
Recebe : nada
Devolve : valor lógico |
| oTopo() → | devolve o objeto do topo da pilha, sem desempilhá-lo
Recebe : nada
Devolve : objeto genérico |

Vamos primeiramente definir uma interface genérica, que apresente os métodos básicos independentemente do tipo de armazenamento escolhido (se vetor ou lista). Isso cria uma "camada de isolamento" entre a implementação da estrutura de dados e o programa de aplicação que utiliza a pilha, de maneira que os detalhes da implementação internos ficam desconhecidos da aplicação, e apenas os nomes de métodos são conhecidos e acessíveis. A nossa interface não especificará que tipos de objetos podem ser armazenados na pilha, de modo que qualquer objeto poderá ser armazenado na pilha. Abaixo temos a descrição da interface:

```
interface IStack<Dado>           // Dado indica que trataremos classes genéricas na pilha
{
    void Empilhar(Dado elemento);   // empilha elemento, que é da classe genérica Dado
    Dado Desempilhar();            // desempilha e retorna o elemento da classe Dado
                                    // que está no topo da pilha
    Dado OTopo();                 // retorna o elemento do topo da pilha sem removê-lo
    int Tamanho { get; }          // propriedade informa quantos Dados empilhados
```

```
bool EstaVazia { get; }           // propriedade informa se a pilha está ou não vazia
}
```

Quando ocorrer de se buscar o elemento do topo da pilha e esta estiver vazia, deve-se disparar uma exceção, que cuidará dessa situação de erro. Essa exceção é definida pelo código abaixo:

```
class PilhaVaziaException : Exception
{
    public PilhaVaziaException(string mensagem) : base(mensagem)
    {
    }
}
```

Após termos definido a classe abstrata, que informa os métodos que realizam as operações de uma pilha, devemos definir uma classe concreta que implemente os métodos acima. Essa implementação pode ser feita de várias maneiras, e o importante aqui é notar que, qualquer que seja a maneira de implementar essas operações internamente, a interface com a aplicação continua sendo aquela definida acima. Isso permite que a aplicação funcione normalmente mesmo que a implementação da estrutura de dados seja modificada.

Por exemplo, podemos notar que uma estrutura de dados compatível com a definição de pilha é um objeto que contenha o **apontador do Topo** e a **área de armazenamento**, que pode ser um vetor. Pode-se usar outras estruturas para a área de armazenamento, como uma lista ligada. Mas, para efeito de simplicidade, neste momento usaremos um vetor.

Um vetor possui certas características que facilitam a implementação, pois torna-a muito simples. No entanto, um vetor tem que possuir um número de elementos previamente definido quando é criado, de modo que quando começamos a implementar a pilha usando um vetor, precisamos definir seu tamanho N. Assim, nossa pilha será composta por um **vetor P de N posições** e uma variável **topo** que informará o índice do elemento do topo do vetor P dessa pilha.

Em C# e Java, os índices de um vetor começam com o valor 0. Portanto, iniciaremos topo com -1 e usamos esse valor para determinar se a pilha está vazia e também para sabermos quantos elementos a pilha possui a cada momento.

A implementação com vetor causa um problema que não ocorreria com listas ligadas: o problema de “lotação” esgotada na pilha, uma vez que o vetor P tem um número fixo de posições. Caso se necessite empilhar mais valores do que o tamanho do vetor P, então o objeto Pilha deverá disparar uma exceção PilhaCheiaException indicando esse erro. Numa implementação com listas ligadas essa exceção não existiria, como veremos oportunamente no nosso estudo de listas ligadas.

Vamos, portanto, pensar nos algoritmo de cada método ou propriedade:

```
Empilhar(o)
    se (tamanho == N) então
        disparar uma PilhaCheiaException
    topo ← topo + 1
    P[topo] ← o

Desempilhar()
    se (estaVazia())
        disparar uma PilhaVaziaException
    o ← P[topo]
    P[topo] ← null
    topo ← topo -1
    retorno o

OTopo()
```

```
    se (estaVazia()) então
        disparar uma PilhaVaziaException
    retorno P[topo]

Tamanho()
    retorno topo + 1

EstaVazia()
    retorno (topo < 0)
```

Note que no último método, desempilhar(), atribuímos **null** a P[topo] logo após copiarmos o valor empilhado no objeto genérico **o**. Isso é feito para que o mecanismo de coleta de lixo (collect garbage) de C# recupere a área de memória não mais utilizada e a devolva para o sistema operacional, de modo que outros processos possam utilizá-la.

Em seguida, vamos criar a classe concreta que implementa a interface IStack utilizando os algoritmos acima, com vetores:

```
class PilhaVetor<Dados> : IStack<Dados>
{
    int maximoPosicoes;
    Dado[] p; // vetor onde serão armazenados os dados empilhados
    int topo; // índice da posição usada por último nesse vetor

    public PilhaVetor(int posic)
    {
        p = new Dado[posic];
        maximoPosicoes = posic;
        topo = -1;
    }

    public PilhaVetor() : this(500)
    { }

    public void Empilhar(Dado elemento)
    {
        if (topo == maximoPosicoes)
            throw new Exception("Pilha transbordou!");
        p[++topo] = elemento;
    }

    public Dado Desempilhar()
    {
        if (EstaVazia())
            throw new PilhaVaziaException("Pilha esvaziou!");
        var valor = p[topo--];
        return valor;
    }

    public Dado OTopo()
    {
        if (EstaVazia())
            throw new PilhaVaziaException("Pilha esvaziou!");
        return p[topo]; // devolve o valor armazenado na última posição em uso do vetor p
    }

    public int Tamanho { get => topo+1; }
    public bool EstaVazia { get => topo < 0; }
}
```

Abaixo temos a classe que implementa a exceção PilhaCheiaException:

```
class PilhaCheiaException : Exception
{
    public PilhaCheiaException(string mensagem) : base(mensagem)
    {
    }
}
```

Falta, agora, estudarmos como utilizar a nossa pilha genérica para armazenar objetos de um tipo específico de dados. Como você pôde notar, a forma como definimos a pilha permite que ela armazene qualquer tipo de objeto.

3. Uma primeira aplicação

Sempre que houver necessidade de se guardar elementos para uso posterior, na ordem inversa à que foram guardados, pode-se usar uma pilha como estrutura de armazenagem. Como exemplos de aplicações de pilha estudaremos o balanceamento de parênteses e algoritmos de backtracking.

Balanceamento de Parênteses

Suponha que você tem uma sequência de caracteres do tipo {}, [(),], {}, e necessite de um algoritmo que verifique se a sequência está ou não balanceada, ou seja, se os símbolos acima se combinam (casam) na sequência dada. Exemplo:

Balanceada	Não-balanceada
vazia	(()] (
()]) []
[([]) () { () }]	[{ () }]

Pode-se fazer o algoritmo de duas formas, pelo menos.

A primeira solução consiste em ler toda a sequência, e então percorrê-la várias vezes. A cada vez, tentar "casar" (combinar) dois símbolos correspondentes ((e), [e], { e }), retirando-os da sequência, até que esta fique vazia. Se sobrar algum símbolo sem seu correspondente, a sequência original não estava平衡ada. Por exemplo, suponha que a sequência abaixo foi lida e armazenada num vetor:

```
{ ( ( ( { } [ ( ) ] { ( ) } ) ( ) ] ) { ( ( ) ) } }
```

Após uma verificação visual, sabe-se que é balanceada.

A aplicação do algoritmo acima descrito em linhas gerais, resultaria no seguinte:

```
{ ( ( ( { } [ ( ) ] { ( ) } ) ( ) ] ) { ( ( ) ) } } →
  ↗ casou ( retira da sequência )
{ ( ( ( [ ( ) ] { ( ) } ) ( ) ] ) { ( ( ) ) } } →
{ ( ( ( ( ( ) ) ( ) ] ) { ( ( ) ) } } →
  ↗
{ ( ( ( ( ) ) ( ) ] ) { ( ( ) ) } } →
```

```

{ ([ () ]) { ( () ) } } → { ([ () ]) { ( () ) } } →
{ ([ ] ) { ( () ) } } → { ( ) { ( () ) } } →
{ { ( () ) } } → { { ( ) } } →
{ { } } → { } vazia! Logo, balanceada.

```

A sequência inicial tinha 26 símbolos. Foram necessários 13 iterações para reduzí-la a uma sequênciá vazia, que, **por definição**, é balanceada.

No entanto, foi necessário ler toda a sequência antes de processá-la. Assim, se houver N elementos, serão necessárias N leituras, e mais N/2 repetições do balanceamento para detectar se a sequência está ou não balanceada.

Nessa solução, usa-se um vetor para armazenar a sequência original. A cada combinação de símbolos, os símbolos à direita devem ser deslocados para a esquerda para serem sobrepostos aos que casaram.

Isto também leva tempo. E, para piorar, é necessário reiniciar as comparações desde o primeiro símbolo da sequência, para detectar a próxima combinação.

Para se diminuir o tempo gasto nesta solução, pode-se usar uma pilha. Esta é a segunda solução, que consiste em, ao mesmo tempo da leitura, ir-se empilhando os símbolos de abertura ({, (, e [). Quando o símbolo lido for um de fechamento (),),], desempilha-se o último de abertura. Se este for correspondente ao de fechamento, continua-se a leitura e as comparações, até que o final da sequência seja encontrado. Se, neste momento, a pilha estiver vazia (TOPO = -1), a sequência é balanceada. Caso fique algum símbolo na pilha, significa que não se encontrou seu par correspondente, e, portanto, a sequência era não-balanceada.

Este processo gasta, em termos de tempo, o tempo para se ler, comparar e fazer operações com a pilha para os N elementos da sequência. Isto é bem menos que o gasto na primeira solução, como mostra a tabela abaixo, realizada a partir de um estudo feito num PC AT 286 com 16 MHz de clock:

	Caso 1	Caso 2	Caso 3
N	90	598	598
Método 1	0.06	0.60	0.60
Método 2	0.05	0.11	0.06

Caso 1 - 90 caracteres, com a função de balanceamento chamando Empilha e Desempilha

Caso 2 - 598 caracteres, nas condições acima

Caso 3 - 598 caracteres, com os comandos internos aos procedimentos Empilha e Desempilha escritos dentro da função de balanceamento (economiza tempo de chamada)

As sequências usadas estavam balanceadas, para que o método usado não terminasse ao detectar não-balanceamento. Assim, percorreu-se toda a sequência.

O método 2 também economiza memória, pois no primeiro método é necessário armazenar-se, na pior das hipóteses, todos os N elementos da sequência num vetor. O mesmo se dá na melhor hipótese e na hipótese média.

colchetes e chaves digitada no TextBox.

O correspondente ao algoritmo discutido acima na linguagem C# é simples. Suponhamos que pilha é uma instância de PilhaVetor, onde o objeto a ser empilhado seja de tipo **string** e a função Combinam(A,B) verifica se os dois símbolos A e B são correspondentes (por exemplo, } e {).

```
private void btnAnalizar_Click(object sender, EventArgs e)
{
    Analisar();
}

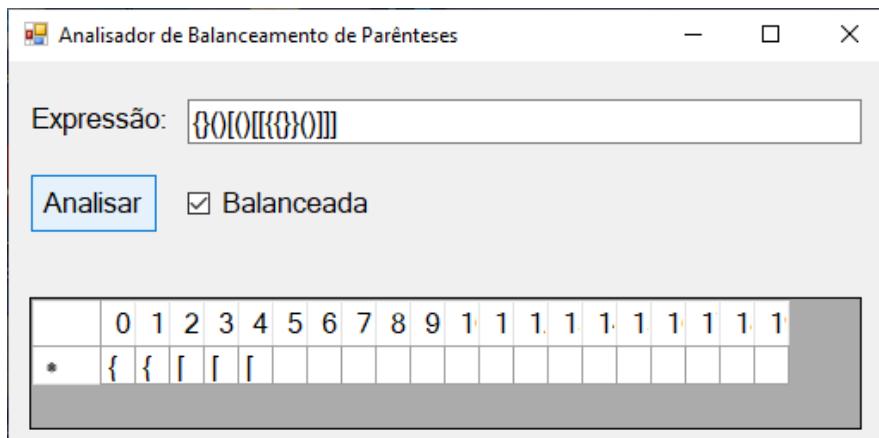
void Analisar()
{
    bool balanceada = true;
    string cadeia = txtExpressao.Text;
    PilhaVetor<String> pilha = new PilhaVetor<String>();
    for (int i = 0; i < cadeia.Length && balanceada; i++)
    {
        if (EhAbre(cadeia[i] + ""))
            pilha.Empilhar(cadeia[i] + "");
        else
        {
            string caracterAbertura = pilha.Desempilhar();
            if (!Combinam(caracterAbertura, cadeia[i] + ""))
                balanceada = false;
        }
        pilha.Exibir(dgvPilha);
    }
    if (!pilha.EstaVazia)
        balanceada = false;
    chkBalanceada.Checked = balanceada;
}

public bool EhAbre(string caracter)
{
    return (caracter == "{" || caracter == "[" || caracter == "(");
}
private bool Combinam(string abre, string fecha)
{
    return (abre == "{" && fecha == "}") ||
           (abre == "[" && fecha == "]") ||
           (abre == "(" && fecha == ")");
}

private void txtExpressao_TextChanged(object sender, EventArgs e)
{
    chkBalanceada.Checked = false;
}
```

A aplicação acima recebe uma sequência e verifica se ela é ou não balanceada, devolvendo **True** ou **False**, respectivamente.

Esse algoritmo pode ser aplicado em compiladores para testar balanceamento de expressões parentizadas. Pode ser usado tam-



bém para detectar erros de sintaxe em programas fonte, tais como **Begin** casando com **End** em programas escritos em Pascal, Algol ou PL/I ou mesmo chaves num programa escrito em C, C++ ou Java.

Na classe PilhaVetor, codificamos o método abaixo para exibir a Pilha no DataGridView. Analise-o e perceba que não percorremos a pilha como um vetor mas, ao contrário, usamos a disciplina LIFO para percorrê-lo, gerando uma pilha auxiliar para colocar os dados na ordem correta de exibição e, posteriormente, recoloca-los na pilha original, de modo que o programa de aplicação não perceba nenhuma modificação na pilha:

```
public void Exibir(DataGridView dgv)
{
    for (int j = 0; j < 20; j++)
        dgv[j, 0].Value = "";

    PilhaVetor<Dado> auxiliar = new PilhaVetor<Dado>();
    int i = 0;
    while (!this.EstaVazia)
    {
        dgv[i++, 0].Value = this.OTopo();
        Thread.Sleep(300);
        Application.DoEvents();
        auxiliar.Empilhar(this.Desempilhar());
    }
    while (!auxiliar.EstaVazia)
        this.Empilhar(auxiliar.Desempilhar());
}
```

4. Pilha baseada em Lista Ligada

Você poderá ter notado que a inclusão de elementos na lista como fizemos em nosso estudo anterior parece-se muito com a operação de empilhamento de elementos em uma pilha.

Portanto, uma lista ligada pode ser usada para implementar uma pilha, com a vantagem de não se ter de alocar o espaço de um vetor antecipadamente, e correr o risco de esse espaço ser pequeno demais para a quantidade de valores a empilhar, ou de ter-se desperdício de memória no caso de termos poucos valores a empilhar.

Para usarmos a estrutura de dados Pilha baseada em uma lista ligada, vamos criar uma nova implementação da interface Stack, a classe PilhaLista1:

```
public class PilhaLista1<Dado> : IStack<Dado> where Dado: IComparable<Dado>
{
    NoLista<Dado> topo;
    int tamanho;

    public PilhaLista1()           // construtor
    {
        topo = null;
        tamanho = 0;
    }

    public int Tamanho { get => tamanho; }

    public bool EstaVazia { get => topo == null; }

    public void Empilhar(Dado o)
    {
        // Instancia um nó, coloca o Dado o nele e o liga ao antigo topo da pilha
        NoLista<Dado> novoNo = new NoLista<Dado>(o, topo);
```

```

        topo = novoNo;           // topo passa a apontar o novo nó
        tamanho++;                // atualiza número de elementos na pilha
    }

    public Dado OTopo()
    {
        if (EstaVazia)
            throw new PilhaVaziaException("Underflow da pilha");
        return topo.Info;
    }

    public Dado Desempilhar()
    {
        if (EstaVazia)
            throw new PilhaVaziaException("Underflow da pilha");
        Dado o = topo.Info;      // obtém o objeto do topo
        topo = topo.Prox;        // avança topo para o nó seguinte
        tamanho--;                // atualiza número de elementos na pilha
        return o;                  // devolve o objeto que estava no topo
    }
}

```

Note que nesta implementação não usamos a classe `ListaSimples`, mas poderíamos tê-lo feito sem problemas, embora não tenhamos necessidade de um apontador para o último elemento, numa pilha.

O método `Empilhar()` é bastante parecido, em termos lógicos, com o método `InserirAntesDoInicio()` da classe `ListaSimples`. Note que a implementação de `Empilhar()` não dispara uma exceção de Pilha Cheia, uma vez que não usamos vetores, que possuem tamanho fixo. O controle da existência de memória disponível fica a cargo do gerenciador de memória do .Net Framework ou da JVM do Java. Caso não exista memória disponível, uma exceção do tipo de `OutOfMemoryException` será disparada.

A figura a seguir mostra como ficaria nossa pilha após três empilhamentos:

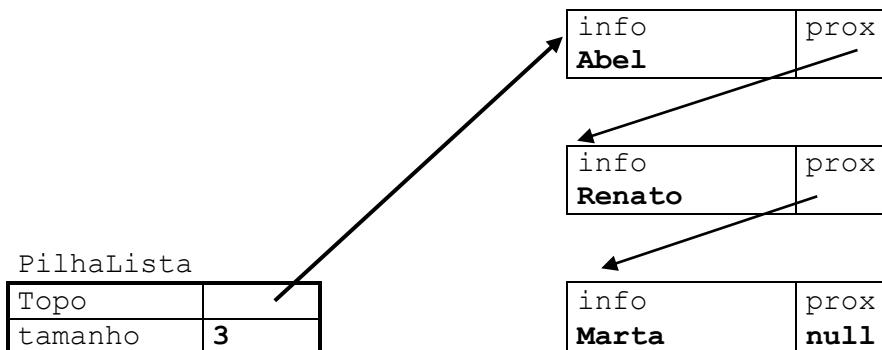


Figura 28 – uma pilha, com três elementos empilhados

No desempilhamento do primeiro elemento da pilha mostrada acima, teríamos que topo passaria a apontar para o segundo elemento, após devolver na lista de parâmetros o objeto que era o antigo topo. Vemos essa configuração na figura abaixo:

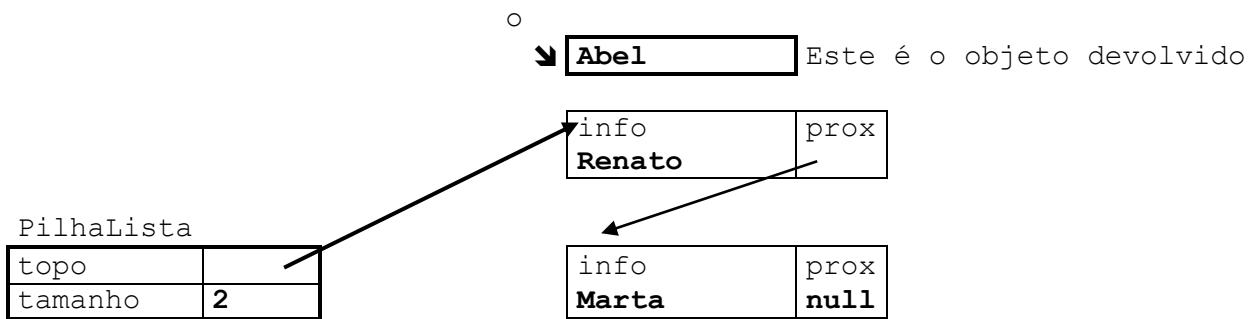


Figura 29 – uma pilha, após desempilhar um elemento

Note que a interface dos métodos continua a mesma, quando comparamos PilhaVetor e PilhaLista. Isso significa que não precisaremos modificar os comandos das aplicações que já temos e que utilizam invocam os métodos de pilha, caso desejemos passar a usar a pilha baseada em lista ligada ao invés da baseada em vetor.

Criando a pilha através da herança de ListaSimples

```
class PilhaLista<Dado> : ListaSimples<Dado>, IStack<Dado> where Dado: Icomparable<Dado>
{
    public Dado Desempilhar()
    {
        if (EstaVazia)
            throw new PilhaVaziaException("pilha vazia!");

        Dado valor = base.Primeiro.Info;

        NoLista<Dado> pri = base.Primeiro;
        NoLista<Dado> ant = null;
        base.removerNo(ref pri, ref ant);
        return valor;
    }

    public void Empilhar(Dado elemento)
    {
        base.InserirAntesDoInicio(new NoLista<Dado>(elemento, null));
    }

    new public bool EstaVazia { get => base.estaVazia; }

    public Dado OTopo()
    {
        if (EstaVazia)
            throw new PilhaVaziaException("pilha vazia!");

        return base.Primeiro.Info;
    }

    public int Tamanho { get => base.QuantosNos; }
}
```

Na classe ListaSimples, criaremos o método RemoverPrimeiro(), que removerá o primeiro elemento da lista ligada e atualizará os demais ponteiros corretamente.

```
public NoLista<Dado> RemoverPrimeiro()
{
    if (EstaVazia)
        throw new Exception("lista vazia");
```

```
NoLista<Dados> devolvido = Primeiro; // primeiro é um NoLista
Primeiro = Primeiro.Prox; // avança o ponteiro para o 2º nó
if (Primeiro == null) // se, ao remover o 1º nó, a lista ficou vazia, então
    ultimo = null; // atualizamos o ponteiro ultimo para não apontar nenhum nó
    quantosNos--;
return devolvido;
}
```

5. Outra Aplicação: Conversão e Resolução de Expressões Aritméticas

Uma expressão aritmética é um conjunto de **operandos** e **operadores**, como os exemplos abaixo:

- 1) A
- 2) A + B
- 3) A – B * C ↑ D + E / F
- 4) C – A / E + B

Os operandos, nos exemplos acima, são representados por letras. Os operadores são os sinais das operações.

Normalmente, os operadores são binários, ou seja, eles atuam sobre dois operandos. Dessa maneira, pode-se dizer que uma expressão é composta por duas sub-expressões mais simples, unidas por meio de um operador binário. Por sua vez, cada sub-expressão pode ser também composta por outras sub-expressões, unidas por outros operadores, e assim por diante, até que se chegue à expressão mais simples de todas, formada por apenas um operando, como a expressão 1 da lista acima.

A expressão 1 e 2 são as mais simples. Mas as duas seguintes podem conduzir a ambiguidades. Tanto se pode imaginar a sub-expressão A-B*C/D+E/F como significando A menos o resultado de B vezes C dividido por D somado a E dividido por F, quanto A menos B, multiplicado por C dividido por D mais E, e tudo isso dividido por F.

Para evitar as ambiguidades, é atribuído a cada operador uma precedência, ou prioridade, sobre os demais. Assim,

Operador	Precedência
↑	1
* , /	2
+ , -	3

ou seja, a operação de potenciação tem precedência sobre as demais, e as operações de soma e subtração têm precedência igual entre si, e em relação às outras três, são as menos prioritárias.

Assim, as expressões 1 a 4, dadas acima, significam o seguinte:

- 1) O valor de A
- 2) A somado a B
- 3) C elevado a D, multiplicado por B, somado a E dividido por F. Tudo isso é subtraído de A
- 4) A dividido por E, somado a B, tudo isso subtraído de C

Mas, e se quiséssemos alterar estes resultados, ou melhor, se fosse necessário mudar a ordem ditada pelas precedências? Tertão, seria preciso usar parênteses. Logo, os parênteses servem para alterar as precedências. Portanto, temos nova tabela de precedências:

Operador	Precedência
(1
↑	2
* , /	3
+ , -	4

Portanto, poderíamos ter as expressões

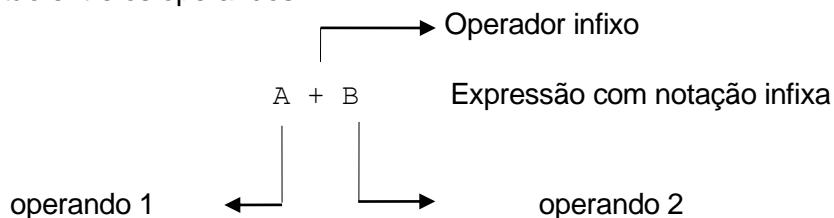
$$3) (A - (B * (C \uparrow D)) + (E / F)) \text{ (Idem à anterior)}$$

ou

$$(((A - B) * C) \uparrow D + E) / F$$

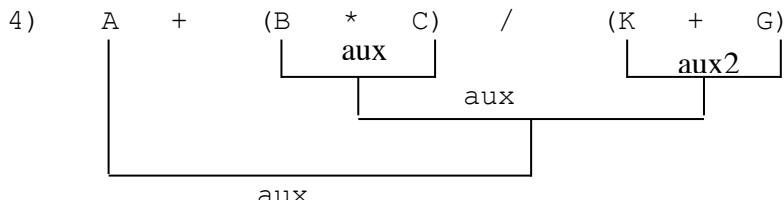
diferindo no resultado final, devido ao uso de parênteses em diferentes posições da expressão.

As expressões apresentadas são chamadas de expressões de **notação infix**, pois os operadores estão entre os operandos.



Quando se começou a escrever os primeiros compiladores para linguagens de alto nível, a avaliação de expressões aritméticas trouxe uma série de problemas de solução difícil. Era muito difícil solucionar uma expressão como a do 3º e 4º exemplos. Isto dificultou bastante a escrita de comandos aritméticos de muitas linguagens, como COBOL, por exemplo.

Em COBOL, os comandos aritméticos, em geral, só permitem o uso de um operador. Logo, as expressões mais complexas, com vários operadores, nem sempre com a precedência normal, devem ser feitas por partes. Por exemplo, a expressão aritmética abaixo seria escrita, em COBOL, com os comandos após a expressão:



Mulitply B by C Giving Aux

Add K, G Giving Aux2

Divide Aux2 Into Aux

Add A to Aux

Assim, havia uma restrição forte às expressões aritméticas complexas, que tinham que ser escritas em várias partes separadas. Isto facilitava o trabalho do compilador, mas complicava a parte do programador. Note como o programador tinha de decompor a expressão em suas sub-expressões básicas, e calculá-las separadamente, antes de chegar ao resultado final.

O uso de expressões infixas com parênteses complica ainda mais a análise do compilador.

Em 1951, um matemático polonês chamado Jan Lukasiewicz, desenvolveu o conceito de expressões aritméticas que não necessitam de parênteses para retirar ambiguidades. Este conceito é o que se passou a chamar de **notação Polonesa** ou **notação prefixa**. A partir dela desenvolveu-se outro tipo de notação, a **polonesa reversa** ou **pósfixa**, que é usada para os cálculos em calculadoras Hewlett Packard.

As expressões que usamos como exemplo seriam, em notação pósfixa, escritas da seguinte maneira:

Infixa	Pósfixa
1) A	A
2) A + B	AB+
3) A - B * C ↑ D + E / F	ABCD↑*-EF/+
4) C - A / E + B	CAE/-B+
5) A + (B * C) / (K + G)	ABC*KG+/+

A notação pósfixa permitiu a resolução dos problemas que ocorriam na avaliação de expressões aritméticas. Bastou criar um algoritmo que convertesse uma expressão infixa em pósfixa, e outro de avaliação da expressão pósfixa, e isso facilitou sobremaneira o problema; assim, melhoraram-se vários compiladores, e mesmo a linguagem COBOL passou a dispor do comando COMPUTE, que permite o uso de expressões infixas complexas e parentizadas.

Compute aux = A + (B * C) / (K + G)

O compilador da linguagem analisa a expressão e, por meio de métodos semelhantes ao que estudaremos a seguir, converte as diversas partes da expressão em seqüências pósfixas, que são posteriormente convertidas para código de máquina que faça os cálculos de duas em duas sub-expressões, maneira que se adapta à arquitetura dos computadores modernos.

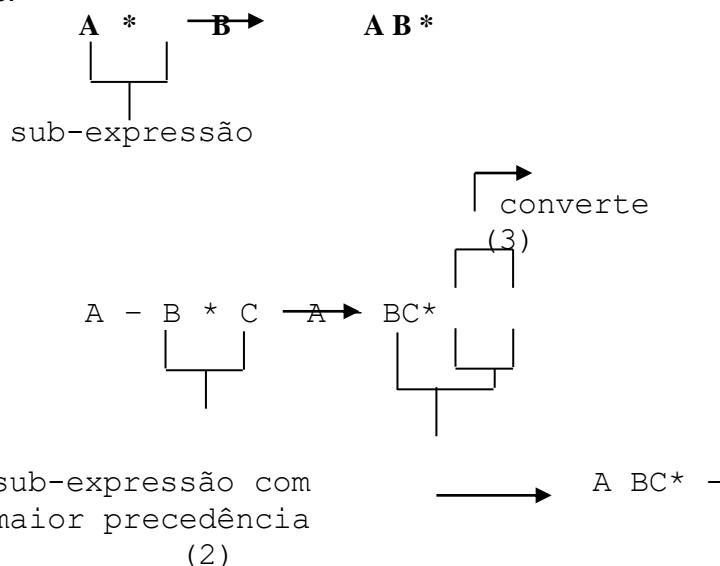
Hoje existem linguagens que, por definição, são pósfixadas, como **FORTH**, tanto para as expressões aritméticas como para comandos. Isto retira do compilador o algoritmo de conversão infixa-pósfixa, tornando-o menor e mais rápido, embora aumente o tempo gasto pelo programador antes que ele se acostume a essa maneira de escrever, pois a lógica de programação normalmente continua a mesma, mudando apenas a sintaxe dos comandos e expressões.

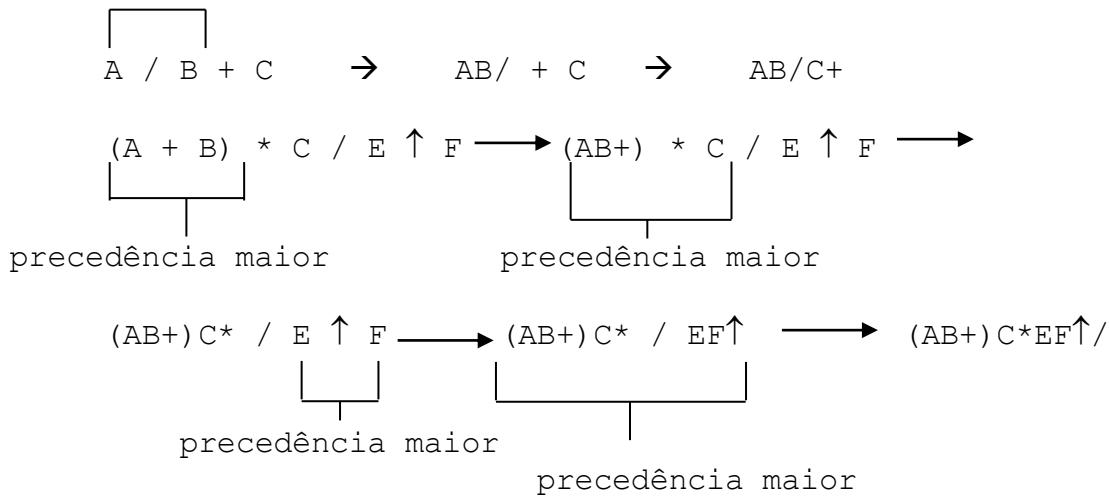
Conversão de Infixa para Pósfixa

Para desenvolvermos um algoritmo de conversão de notação infixa para pósfixa, observemos os passos a seguir:

1. verificar na sequência as sub-expressões que a formam
2. identificar a sub-expressão com maior precedência que ainda não tenha sido convertida; será algo do tipo A | B, onde A e B são operandos e | um operador binário.
3. Efetuar a conversão, colocando o operador após os dois operandos : AB|. Esse resultado passa a ser visto agora como um único operando.
4. Repetir as operações 2 e 3 até que se tenha terminado a conversão.

Exemplo:





Por fim, tirar os parênteses : $AB+C^*EF\uparrow/$

Os parênteses não são necessários, pois agora os operadores aparecem na ordem de execução do cálculo, não sendo mais preciso mudar essa ordem.

Quando dois operadores de mesma precedência aparecerem em sequência, deve-se converter primeiro a sub-expressão mais à esquerda, já que esta apareceu antes que a outra (portanto, tem precedência sobre a outra). No entanto, **quando os operadores em sequência forem ↑, tem maior precedência a sub-expressão à direita**. Exemplos:

$$A + B + C \xrightarrow{AB+} + C \xrightarrow{AB+C+}$$

$$A \uparrow_1 B \uparrow_2 C \longrightarrow A \uparrow_1 BC \uparrow_2 \longrightarrow ABC \uparrow_2 \uparrow_1$$

$$\begin{array}{ll} A / B \uparrow C & ABC \uparrow / \\ A * B * C & AB * C * \\ (A + B) / C * D & AB + C / D * \\ (A + B) / (C * D) & AB + CD * / \end{array}$$

$$A + B \uparrow C / (D \uparrow E * F) / G / H$$

$$A - B / (C * D \uparrow E)$$

$$ABC \uparrow DE \uparrow F * / G / H / +$$

$$ABCDE \uparrow * / -$$

$$\begin{aligned} ((A-B) / C - (D+E)) \uparrow F \uparrow G &\xrightarrow{\quad AB- / C - DE+ \quad} FG \uparrow \\ (AB-C / - DE+) FG \uparrow \uparrow &\xrightarrow{\quad AB-C / DE+ - FG \uparrow \uparrow \quad} \end{aligned}$$

Algoritmo de conversão

Converter $(A \uparrow B * C - D + E / F / (G + H))$
 4 3 2 1 1 2 2 4 1 00

Os números abaixo da expressão indicam a prioridade (precedência) de cada operador.

Como se nota nos exemplos, a ordem de aparecimento dos operandos em ambas as sequências é a mesma. Logo, ao se fazer o algoritmo de conversão, deve-se escrever os operandos na sequência pósfixa à medida que são lidos da sequência infixa.

E os operadores? Para eles, devemos posфиксar primeiro os mais prioritários; devemos ir **guardando-os na pilha** até encontrar um outro com **prioridade maior no topo da pilha**. Quando isso acontecer, devemos colocar na sequência pósfixa os operadores guardados (desempilhando-os), **até o topo chegar a um operador de menor precedência** que aquele operador lido, ou a pilha ficar vazia.

A regra portanto, é **guardar os operadores de maior precedência na pilha**, até aparecer um de menor precedência que o topo, quando então deve-se descarregar a pilha até aparecer um operador de menor precedência que o lido, que então deve ser empilhado para uso futuro.

Assim, para a expressão acima, teremos :

Entrada Infixa	Pilha	Sequência Pósfixa
$(A \uparrow B * C - D + E / F / (G + H))$	(
$A \uparrow B * C - D + E / E / (G + H))$	(A
$\uparrow B * C - D + E / F / (G + H))$	(\uparrow)	A
$B * C - D + E / F / (G + H))$	(\uparrow)	AB
$* C - D + E / F / (G + H))$	(*)	AB \uparrow
$C - D + E / F / (G + H))$	(*)	AB \uparrow C
$- D + E / F / (G + H))$	(-)	AB \uparrow C*
$D + E / F / (G + H))$	(-)	AB \uparrow C*D
$+ E / F / (G + H))$	(+)	AB \uparrow C*D-
$E / F / (G + H))$	(+)	AB \uparrow C*D-E
$/ F / (G + H))$	(+/)	AB \uparrow C*D-E
$F / (G + H))$	(+/-)	AB \uparrow C*D-EF
$/ (G + H))$	(+/-)	AB \uparrow C*D-EF/
$(G + H))$	(+/- (AB \uparrow C*D-EF/
$G + H))$	(+/- (AB \uparrow C*D-EF/G
$+ H))$	(+/- (+	AB \uparrow C*D-EF/G
$H))$	(+/- (+	AB \uparrow C*D-EF/GH
$))$	(+/- (+	AB \uparrow C*D-EF/GH+
$))$	(+/- (+	AB \uparrow C*D-EF/GH+/-

Para implementarmos o algoritmo, vamos precisar definir bem as prioridades dos operadores. Para isso, pode-se montar uma tabela de precedência, como a que se segue; ela será usada para determinar se o operador lido deve ou não ser desempilhado:

	(↑	*	/	+	-)	Símbolo Lido
Símbolo na Pilha (apareceu antes na sequência)	(F	F	F	F	F	F	(último lido)
	↑	F	F	T	T	T	T	
	*	F	F	T	T	T	T	
	/	F	F	T	T	T	T	
	+	F	F	F	F	T	T	
	-	F	F	F	F	T	T	
)	F	F	F	F	F	F	

Por exemplo, se **lemos** um símbolo ‘-’, e há no topo da pilha um ‘(‘, este não tem precedência sobre o ‘-‘, e portanto ‘-‘ deve ser empilhado.

Se foi lido o símbolo ‘(‘, e há qualquer outro no topo da pilha, o símbolo lido deverá ser empilhado, pois não tem precedência sobre nenhum outro.

No entanto, se lemos um ‘*’ e há um ‘/’ no topo da pilha, embora na matemática tenham a mesma precedência, o que apareceu primeiro deve ser desempilhado e colocado na sequência pósfixa. Como ‘/’ apareceu antes na sequência infixa ele tem prioridade sobre ‘*’ deverá ser desempilhado para aparecer na sequência pósfixa antes do ‘*’, que, por sua vez, será empilhado antes de um elemento com precedência menor.

O procedimento relativo ao algoritmo, em C#, vem a seguir:

```
string ConverterInfixaParaPosfixa(string cadeiaLida)
{
    string resultado = "";
    umaPilha = new PilhaLista()           // Instancia a Pilha
    for (int indice=0; indice< cadeiaLida.Length; indice++)
    {
        char simboloLido = cadeiaLida[indice];
        if (!Ehoperador(simboloLido) // not In [‘(‘,’)’,,’+’,,’-’,,’*’,,’/’,,’↑’]
            resultado += simboloLido;      // escreve Operando na saída
        else // operador
        {
            bool parar = false;
            while (! Parar && ! umaPilha.EstaVazia &&
                   TerPrecedencia(umaPilha.oTopo(), Simbolo_Lido))
            {
                char operadorComMaiorPrecedencia = umaPilha.Desempilhar();
                if (operadorComMaiorPrecedencia != '(')
                    resultado += operadorComMaiorPrecedencia;
                else
                    parar = true;
            }
            if (simboloLido != ')')
                umaPilha.Empilhar(simboloLido);
            else // fará isso QUANDO o Pilha[TOPO] = ‘(‘
                operadorComMaiorPrecedencia = umaPilha.Desempilhar();
        }
    } // for

    while (!umaPilha.EstaVazia) // Descarrega a Pilha Para a Saída
    {
        operadorComMaiorPrecedencia = umaPilha.Desempilhar();
        if (operadorComMaiorPrecedencia != '(')
            resultado += operadorComMaiorPrecedencia;
    }
}
```

```

    }
    return resultado;
}

```

`TerPrecedencia(A, B)` é uma função que testa a precedência de A sobre B, usando os dados da tabela de precedência acima. A e B são operadores: A é o elemento do topo da pilha e B o último símbolo lido. Ela pode ser implementada com um comando switch/case, sem necessidade de uma matriz, já que os símbolos não estão em sequência na tabela ASCII e isso dificulta a indexação de matriz.

`TerPrecedencia('*', '+')` = **True**, pois se há um '*' empilhado, ao se ler '+', o '*' deve aparecer antes na sequência pósfixa; logo, tem precedência, e portanto, será desempilhado. Exemplo:

Pilha	Seqüência Pósfixa
A * D + E	A
*	AD
+	AD*
+	AD*E
	AD*E+

Cálculo do Valor de Expressão Pósfixa

Para calcular o valor de uma expressão pósfixa, vamos supor que cada operando tem um valor.

A cada operando da expressão pósfixa (ou seja, um valor), faremos um empilhamento. Quando aparecer um operador, desempilharemos os dois últimos elementos colocados na pilha de operandos, e aplicaremos a eles a operação correspondente ao operador. O resultado desta operação será empilhado, e o processo continua até que se chegue ao final da sequência pósfixa. Nesse momento, haverá apenas um elemento na pilha, que será o resultado da expressão.

Como exemplo, suponha que temos a expressão ao lado: AB-C/DE+-FG↑↑

Os valores dos operandos são os seguintes:

A = 23 B = 7 C = 8 D = 4 E = 2 F = 2 G = 2

O processo de cálculo vem descrito abaixo:

Seqüência pósfixa	Pilha	
AB-C/DE+-FG↑↑	vazia	{ empilha valor de A }
B-C/DE+-FG↑↑	23	{ empilha valor de B }
-C/DE+-FG↑↑	23 7	{ calcula valor da subexpressão }
C/DE+-FG↑↑	16	{ empilha valor de C }
/DE+-FG↑↑	16 8	{ calcula valor da subexpressão }
DE+-FG↑↑	2	{ empilha valor de D }
E+-FG↑↑	2 4	{ empilha valor de E }
+-FG↑↑	2 4 2	{ calcula valor da subexpressão }
-FG↑↑	2 6	{ calcula valor da subexpressão }
FG↑↑	-4 2	{ empilha valor de F }
G↑↑	-4 2 2	{ empilha valor de G }
↑↑	-4 2 2	{ calcula valor da subexpressão }
↑	-4 4	{ calcula valor da subexpressão }
	256	

A função que calcula e devolve o valor de uma expressão pósfixa vem a seguir:

```

double ValorDaExpressaoPosfixa(string cadeiaPosfixa)
{
    umaPilha = new PilhaLista<double>;
    for (int atual=0; atual < cadeiaPosfixa.Length; atual++)
    {
        char simbolo= cadeiaPosfixa[atual];
        if (!EhOperador(simbolo)) // É Operando
            umaPilha.Empilhar(ValorDe[simbolo-'A']);
        else
        {
            double operando2 = umaPilha.Desempilhar();
            double operando1 = umaPilha.Desempilhar();
            double valor = ValorDaSubExpressao(operando1, simbolo, operando2);
            umaPilha.empilhar(Valor);
        }
    }
    return umaPilha.Desempilhar();
}

```

Na implementação em c# do método, supôs-se que a sequência pósfixa estava colocada numa cadeia de caracteres. umaPilha é a pilha onde se armazenará os valores parciais das sub-expressões.

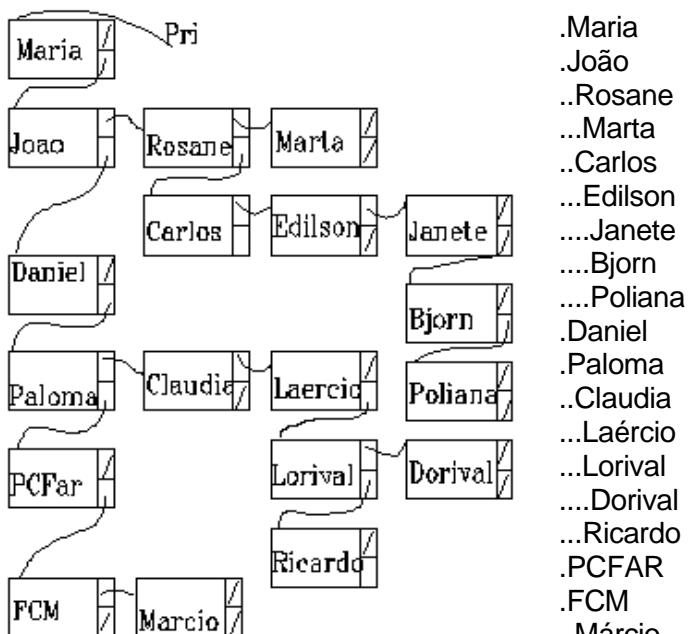
ValorDe é um vetor, indexado por letras de 'A' a 'Z', que contém o valor de cada um dos operandos da expressão. Assim terdo, ValorDe['A'] contém o valor do operando A, ValorDe['B'] contém o valor do operando B e assim sucessivamente.

ValorDaSubExpressao é uma função que calcula o resultado da expressão aritmética simples definida pelos 2 operandos recém-desempilhados e pelo operador retirado da sequência pósfixa.

Como exercício, escreva a função ValorDaSubExpressao em C#.

6. Exercícios

- a. Faça um método que, a partir da lista abaixo, gere a listagem genealógica correspondente:



.Maria
.João
.Rosane
...Marta
..Carlos
...Edilson
....Janete
....Bjorn
....Poliana
.Daniel
.Paloma
..Claudia
...Laércio
...Lorival
....Dorival
...Ricardo
.PCFAR
.FCM
.Márcio

Cada nó é composto pelos campos: Nome, Irmão, e Filho. O apontador para baixo é o apontador Irmão, e o para a direita é o apontador Filho, representado pelo atributo prox.

O método deve percorrer a lista, usando uma pilha para guardar o apontador de retorno para o nó, a cada descendente. Assim, deve-se percorrer, para cada nó, os seus filhos, e após visitar-se todos os filhos, retorna-se e visita-se o irmão do nó desempilhado.

A pilha deve ser implementada com uma lista ligada, sem o uso de vetores.

Figura 31 – uma lista de genealogia

- b. Faça um programa em linguagem C# que leia a descrição de n polinômios ($2 \leq n \leq 9$) de grau m e ordem 1, e calcule polinômios resultantes de operações (parentizadas ou não) de soma, multiplicação e subtração dos polinômios de entrada. Para tanto, use os conceitos de calculadora aprendidos na sessão anterior.
3. Uma das tarefas de um compilador Pascal é verificar se todos os pares **BEGIN-END**, **CASE-END** e **REPEAT-UNTIL** estão corretamente combinados. Faça um programa em Java que leia um arquivo texto, cujo conteúdo é um programa fonte escrito em Pascal, e verifique se este arquivo está correto do ponto de vista da combinação dos pares acima. Ao primeiro erro que for encontrado, o seu programa deve terminar. Para verificar se as palavras se combinam, o seu programa deve ler caracter a caracter o arquivo e montar cada palavra numa **String**. Quando a palavra for **BEGIN**, **REPEAT** ou **CASE**, esta deve ser **registrada** para que se saiba que um **END** ou um **UNTIL** é esperado. Quando um **END** ou um **UNTIL** é encontrado no texto, deve-se liberar a palavra antes registrada, pois já se encontrou o símbolo esperado. É claro que se deve verificar se as duas palavras combinam entre si. Depois de cada balanceamento de um par dessas palavras, deve-se escrever um comentário indicando o número da combinação, como se mostra no exemplo abaixo:

```
Program Teste;
Var X : byte;
Begin
  ReadLn(X);
  Case X Of
    1 : Begin
      Repeat
        Read(X);
        Write(X);
      Until X = 9;
      CASE A OF
        'X' : X := X-1;
      END;
    End;
    2 : WriteLn('Erro');
  End;
  X:= X + 1;
End.
```

```
Program Teste;
Var X : byte;
Begin { 1 }
  ReadLn(X);
  Case { 2 } X Of
    1 : Begin { 3 }
      Repeat { 4 }
        Read(X);
        Write(X);
      Until { 4 } X = 9;
      CASE { 5 } A OF
        'X' : X := X-1;
      END { 5 };
    End { 3 };
    2 : WriteLn('Erro');
  End { 2 };
  X:= X + 1;
End { 1 }.
```

Cada caracter lido deve ser escrito na tela, levando em conta mudanças de linha e o fim do arquivo. Quando um erro for encontrado, este deve ser exibido abaixo da linha do último caracter processado até o momento do erro.

4. Suponha que você tem um computador que tem um único registrador e seis instruções na sua estrutura. Um registrador é um elemento do hardware do processador que armazenará valores e que será usado como operando em operações aritméticas. Em outras palavras, todas as operações aritméticas serão efetuadas tendo o registrador como um dos operandos e uma variável como o outro operando. O resultado da operação fica no registrador.

As instruções são as seguintes:

LD	X –	coloca o conteúdo da variável X no registrador
ST	X –	coloca o conteúdo do registrador na variável X
AD	X –	soma o conteúdo da variável X ao conteúdo do registrador, que armazena o resultado
SB	X –	subtrai o conteúdo da variável X do conteúdo do registrador, que fica com o resultado
ML	X –	multiplica o conteúdo da variável pelo conteúdo do registrador, que recebe o resultado

DV X – divide o conteúdo do registrador pelo conteúdo de X, e o resultado fica no registrador

Escreva um programa que leia uma expressão pósfixa contendo operandos com uma única letra e os operadores +, -, * e /, e imprima uma sequência de instruções para avaliar a expressão e deixar o resultado final no registrador. Use variáveis da forma TEMPn como variáveis temporárias. Por exemplo, a expressão pósfixa ABC*+DE-/ deveria gerar o seguinte resultado impresso:

LD B	; COLOCA B NO REGISTRADOR
ML C	; MULTIPLICA (REGISTRADOR) POR TER
ST TEMP1	; ARMAZENA RESULTADO PARCIAL
LD A	; CARREGA A NO REGISTRADOR
AD TEMP1	; SOMA (REGISTRADOR) COM TEMP1
ST TEMP2	; GUARDA (REGISTRADOR) EM TEMP2
LD D	; CARREGA D NO REGISTRADOR
SB E	; SUBTRAI E DO (REGISTRADOR)
ST TEMP3	; GUARDA RESULTADO PARCIAL EM TEMP3
LD TEMP2	; CARREGA TEMP2 NO REGISTRADOR
DV TEMP3	; DIVIDE REGISTRADOR POR TEMP3
ST TEMP4	; RESULTADO FINAL EM TEMP4

Cada caracter lido deve ser escrito na tela, levando em terta mudanças de linha e o fim do arquivo. Quando um erro for tercontrado, este deve ser exibido abaixo da linha do último caracter processado até o momento do erro.

5. A linguagem de programação FORTH é um interpretador que tem como característica principal o uso de pilhas para armazenar os dados. Existem vários comandos para manipular as pilhas, como segue abaixo:

0<enter>	- empilha o valor 0	ABREVIATURA 
5<enter>	- empilha o valor 5	
1 6<enter>	- empilha os valores 1 e 6	
.<enter>	- desempilha o topo e o exibe	
DUP<enter>	- exibe o topo sem desempilhá-lo	
OVER<enter>	- empilha o segundo elemento da	
ROT<enter>	- empilha o terceiro elemento da	
SWAP<enter>	- troca os 2 primeiros elementos	
DROP<enter>	- destrói o elemento do topo sem	
5 9 -<enter>	- empilha 5 e 9, desempilha-os e efetua a operação, empilha o resultado e o exibe (-4)	pilha pilha entre si exibi-lo
4 6 2 * +<enter>	- empilha 4, 6 e 2, desempilha 2 e 6, efetua a multiplicação, empilha o resultado desta (12) desempilha 12 e 4, faz a soma e empilha o resultado (16) e o exibe	
4 D 3 O . <enter>	- empilha 4, exibe-o (D), empilha o 3, empilha o segundo elemento (4), retira-o e o exibe(.)	
8 7 6 3 R S	- empilha 8, 7, 6 e 3; empilha 7 TER; troca 7 e 3 entre si (S)	
F	- termina a execução (FIM)	

Faça um programa que leia várias cadeias de 80 caracteres (uma por vez), analise cada uma e simule o funcionamento dos comandos acima quando eles estiverem na cadeia. Pode haver mais de um comando na linha, como nos exemplos dados.

Para facilitar, na cadeia só haverá números positivos de 0 a 9, e os comandos só terão as letras das abreviaturas. Cada elemento da linha de comando estará separado por um único espaço em branco.

7. Backtracking

Backtracking é o nome que se dá a uma técnica de resolução de problemas que envolve a chegada a uma solução por meio de aproximações sucessivas e tentativa-e-erro, também chamado por alguns autores de “Busca Heurística”.

A idéia geral desse tipo de técnica é guardar a solução parcial atual, e a partir desta, procurar a primeira solução parcial que se deriva da atual. Se esta nova parcial se mostrar inválida, retorna-se à solução guardada anteriormente, e testa-se a solução parcial seguinte que se deriva da atual. Se, por outro lado, essa solução nos aproxima mais um pouco da solução final do problema, esta passa a ser a solução atual, e repete-se os passos acima. Caso se esgotem todas as soluções parciais possíveis sem se encontrar a solução final, o problema não terá solução com os dados utilizados.

Existem vários exemplos, dois dos quais apresentamos a seguir.

O Problema do Labirinto (ou Pausa para Meditação)

A mitologia grega conta-nos uma estória que se passou na Ilha de Creta. Poseidon, o Deus do Oceano, enviou ao Rei Minos um touro branco que deveria ser sacrificado em honra do deus. No entanto, o touro era tão belo que o Rei recusou-se a sacrificá-lo e trocou-o por um de seu próprio rebanho. Poseidon, como vingança, fez com que Afrodite obrigasse a esposa de Minos, Pasifae, a se apaixonar pelo touro branco, e a pobre Pasifae deu à luz um ser metade homem, metade touro. Foi chamado de Minotauro, e aterrorizava a população de Cnossos, a capital, até que o Rei ordenou ao seu arquiteto e inventor Dédalo que construísse algo que mantivesse o Minotauro fora das vistas humanas para sempre.

Dédalo construiu então um palácio, na beira de um rochedo, e lá encerrou-se o Minotauro. Para que ele nunca saísse de lá, o palácio era repleto de passagens que nunca levavam a lugar nenhum, ou seja, era um labirinto. Com o tempo, o Rei foi aprisionando seus desafetos políticos no labirinto de Creta, como ficou conhecido o palácio, e assim, resolveu seus problemas principais: a oposição e o boizinho que o bondoso Poséidon proporcionara à sua esposa. A cada nove anos, Minos exigia que os atenienses enviassem 7 rapazes e 7 donzelas para serem sacrificados e servirem de alimento ao Minotauro.



<http://historiadom.wordpress.com/2008/08/>



Poséidon percebeu a trama e enviou seu filho mortal Teseu, que também era de Atenas, para matar o Minotauro, que esfomeado, comia a todos os prisioneiros perdidos no labirinto.

Teseu, que não era bobo, passou a se encontrar às escondidas com Ariadne, a filha de Minos e Pasifae (a propósito, Ariadne era uma garota normal). Ao descobrir isso, o Rei mandou Teseu ser levado para o labirinto, para servir de pasto ao Minotauro.



<http://jornale.com.br/wicca/?p=834>

Ariadne, desesperada, tolinha..., deu-lhe um novelo de lã, que Teseu amarrou na entrada do Labirinto e o foi desenrolando até encontrar-se com o bicho, matando-o, coitadinho, de estômago vazio. Então, reenrolando o fio, chegou à entrada, fugiu e levou Ariadne consigo, até a ilha de Naxos, onde a tola foi abandonada e trocada por sua irmã.

Minos descobriu que Dédalo havia acobertado o romance e a fuga dos dois, e, mesmo sem Minotauro, prendeu Dédalo e seu filho Icaro no Labirinto, esperando que eles andassem errantes até a morte. Após muito tempo andando entre as paredes sem o que comer, (talvez tenham feito churrasco do resto do Minotauro), chegaram a uma janela do labirinto, e observaram o céu azul, o mar, a centenas de metros rochedo abaixo, e as aves perto das nuvens. E Dédalo resolveu fabricar asas para ele e o filho, e depois os dois saíram de lá voando.

O resto da história todo mundo conhece: Icaro voou muito perto do Sol e este derretou a cera que prendia as penas das asas, e Icaro precipitou-se no mar, onde morreu. Dédalo fugiu para a Sicília, onde Minos tentou prendê-lo mas acabou sendo morto.

Mas o que interessa realmente para nós é o método usado por Teseu para encontrar o caminho no labirinto. Ele usou um novelo de lã para marcar as passagens pelas quais já tinha passado, e se por acaso encontrasse uma passagem sem saída, bastava voltar para trás e procurar nova saída.¹

Uma outra estória parecida é a estória de João e Maria. Os dois entraram numa floresta, e para não se perderem, deixavam pelo caminho migalhas de pão, para retornar se preciso. Mas vieram aves e comeram as migalhas, de maneira que eles perderam o caminho de volta. Isso mostra a importância de uma estrutura adequada de dados para armazenar o caminho de volta.

Assim, suponha que desejamos desenvolver um algoritmo que descubra o caminho da saída em um labirinto. Para não andarmos em círculos pelo resto da vida, deveremos marcar cada posição em que passarmos, além de guardar também a direção em que fomos. No caso de Teseu, essas duas informações eram guardadas pelo novelo de lã.

Podemos representar o labirinto como uma matriz de caracteres, na qual um espaço em branco significa passagem livre e asteriscos significam parede, como abaixo:

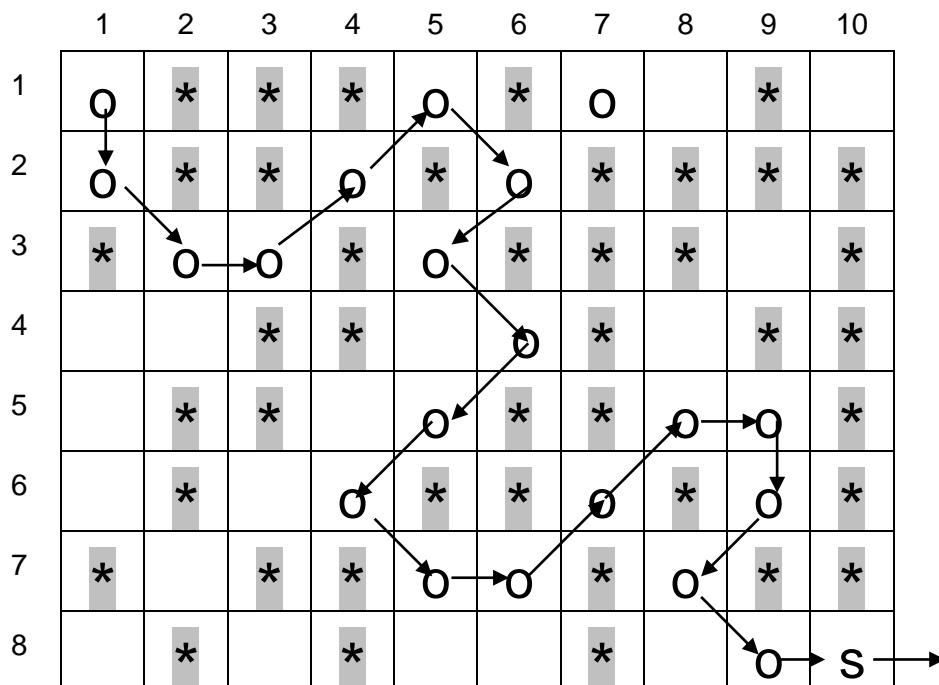


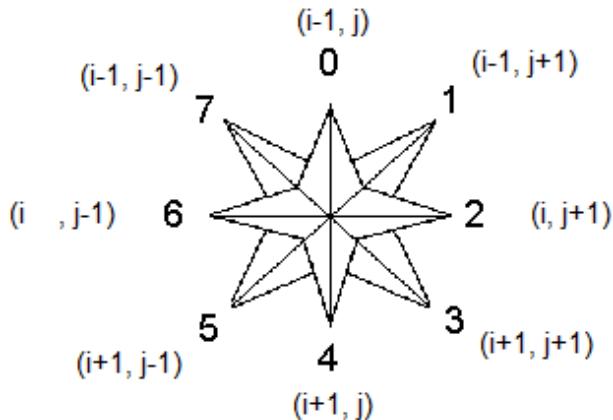
Figura 4 – um labirinto representado por uma matriz e um possível caminho de saída



<http://reinaldorosailustrador.blogspot.com/2011/07/joao-e-maria.html>

¹ Uma ótima descrição da lenda está no site <http://leiturasdahistoria.uol.com.br/ESLH/Edicoes/8/imprime78907.asp>

As setas acima indicam um possível caminho. Pode haver outros, cujos traçados irão depender da direção seguinte escolhida em cada posição que está sendo visitada no momento. De cada posição livre no labirinto, pode haver no máximo 8 direções possíveis de se tomar. Se estamos visitando a posição (i, j) da matriz, as direções possíveis de se caminhar são semelhantes a uma Rosa dos Ventos, como vemos na figura 5, a seguir:



Pode-se colocar estas direções numa tabela de 2 campos para posição, como abaixo:

DIREÇÃO	LIN	COL
0	-1	0
1	-1	1
2	0	1
3	1	1
4	1	0
5	1	-1
6	0	-1
7	-1	-1

Figura 5 – Rosa dos ventos e sua representação vetorial

Se d é o número da direção que se tomará a seguir, e LIN e COL são os campos da tabela acima, para mudarmos da posição (i, j) indo na direção d , basta fazer os seguinte:

$$\begin{aligned} I_{\text{Novo}} &= I + \text{Lin}[d] \\ J_{\text{Novo}} &= J + \text{Col}[d] \end{aligned}$$

Por exemplo, se estamos na posição $(6, 9)$ da matriz e descobrimos que há passagem na direção 5 a partir do ponto em que estamos, para mudar de lugar, basta fazer:

$$\begin{aligned} I &= I + \text{Lin}[d] = 6 + 1 = 7 \\ J &= J + \text{Col}[d] = 9 + (-1) = 8 \end{aligned}$$

Ou seja, passamos de $(6, 9)$ para $(7, 8)$, como vemos na figura a seguir:

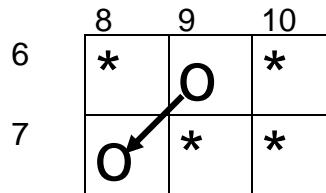


Figura 6 – Mudança de posição no percurso dentro do labirinto

Pois bem, começando na posição $(1, 1)$ desejamos ir à posição (M, N) , onde M e N representam a ordem da matriz. Essa posição será a saída do labirinto. Ou então, podemos convencionar que a saída será a primeira posição do labirinto onde encontrarmos uma letra S. Para chegar à saída, verificaremos se, a partir da posição onde nos encontramos (atual), há passagem em alguma das 8 direções a partir do ponto em que estamos. Quando encontrar uma passagem, mudamos para o ponto seguinte na direção da passagem.

Mas logo de início, a primeira direção produz uma indexação inválida na matriz, para a posição $(0, 1)$. A fim de evitar isto, façamos das bordas da matriz uma parede dupla: em outras palavras, deve-se colocar uma linha a mais em cima e abaixo da matriz, bem como mais uma coluna à direita e à esquerda. Portanto, a matriz deve ser de ordem $(M+1, N+1)$, com bordas preenchidas como paredes.

Após o primeiro movimento, empilha-se a posição anterior e a direção tomada. Como a nova posição não é uma parede, inicia-se o processo novamente, a partir dessa nova posição. É como se estivéssemos acabando de entrar no labirinto novamente, ou seja, se a partida fosse da posição atual.

De certa forma é isso o que ocorre, inclusive porque já se perdeu a lembrança da posição anterior, que está na pilha. Portanto, toma-se agora a direção 0. Supondo que nesta direção há uma parede, tenta-se a direção seguinte (1). Se continuar havendo parede nesta nova direção, tenta-se a seguinte, e assim por diante, até se encontrar passagem, ou então se esgotarem as direções. Nesse caso, estamos num beco sem saída, e portanto, deveremos retornar à posição anterior, que está empilhada, para tentar a próxima direção de saída a partir desse local do labirinto. É como se Teseu encontra-se um local sem saída e reenrolasse o fio de lã para voltar à sala anterior, procurando uma nova saída dessa sala. Se não houvesse mais nenhuma saída, ele enrolaria novamente o fio e voltaria novamente para trás, numa sala ainda mais anterior.

Portanto, sempre que se encontrar uma passagem, empilha-se a posição atual e a direção tomada para a passagem, e caminha-se para a nova posição, iniciando-se a pesquisa de passagem. Sempre que se encontrar num beco sem saída, desempilha-se a posição anterior, e tenta-se prosseguir na direção seguinte àquela que foi desempilhada, partindo da posição anterior também desempilhada.

Pode acontecer de o caminho tomado levar-nos de novo a uma posição já visitada. Assim, corre-se o risco de se andar em círculos, ou seja, do método entrar em LOOP. Para evitar esse risco, podemos usar a própria estrutura do labirinto em nosso favor: devemos marcar cada posição visitada com um símbolo qualquer, diferente de espaço e de parede, para sabermos que já passamos por ali e que devemos procurar outra passagem. Este símbolo, portanto, seria uma espécie de parede, e representa o caso de Teseu encontrar o novelo de lã cruzando algum de seus caminhos. Assim, ele deverá voltar até encontrar outra passagem diferente.

Para iniciar o processo, empilhamos o primeiro movimento ou seja, os dados (1, 1, d), onde (1, 1) é a posição de entrada no labirinto e d é a primeira direção escolhida. A partir daí, enquanto a pilha não esvaziar e não se encontrar a saída, repetimos o processo descrito de pesquisa de caminho, até que se encontre um caminho factível, ou volte-se para o início da matriz, e não haja mais direções a testar a partir deste início. Essa situação indicará que não há saída do labirinto, e a pilha estará vazia.

Grafos Dirigidos e Redes de Caminhos

Um grafo serve para informar a existência de ligação entre vários itens. Esses itens podem ser cidades, estações de uma linha de metrô ou trem, etc. A função do grafo é informar o sentido dessa ligação, se possível, a distância entre eles, ou o tempo ou custo de percurso. Assim, podemos fazer programas que facilitem a tomada de decisões com relação ao melhor percurso, a encontrar todas as ligações entre um item e outro, e até mesmo a descobrir se existe um percurso entre dois itens, passando por outros itens desejados, como o GoogleMaps ou o BingMaps:

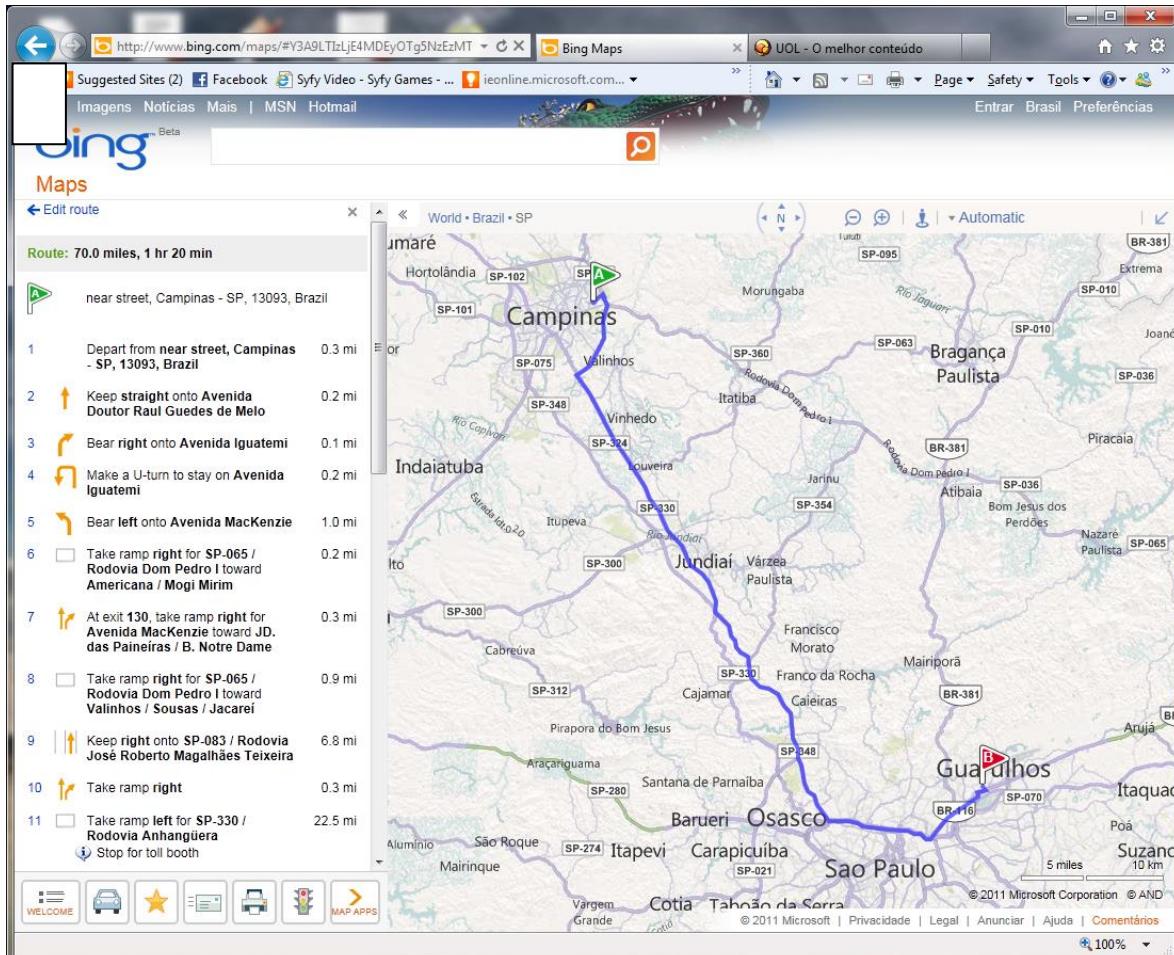


Figura 7 – Apresentação do Bing/Maps, mostrando um possível caminho de Campinas a Guarulhos

Num outro exemplo, a figura 8 mostra um mapa de uma região imaginária com 10 cidades. A partir de cada cidade pode ou não sair um caminho em direção a outra cidade. Caso isso aconteça, existirá uma flecha indicando a direção de saída. Perto da flecha está indicada a distância entre as cidades por aquele caminho. Pode ocorrer também que de uma cidade possa se voltar a outra da qual se saiu, direta ou indiretamente. As cidades 0 e 7 são isoladas, como se pode ver no grafo. As cidades 8 e 9 têm entrada mas não têm saída. Pode-se sair da cidade 1 e chegar-se à cidade 5, passando-se pelas cidades 2, 3 e 4, nessa ordem.

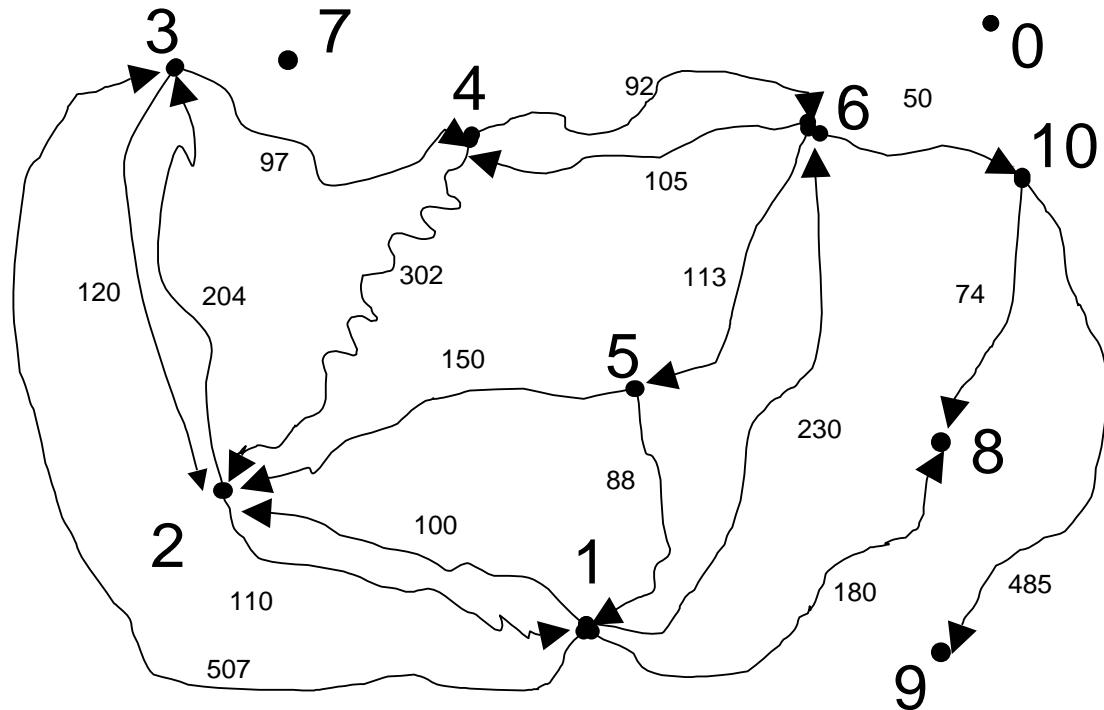


Figura 8 – Um grafo representando caminhos entre cidades e o comprimento de cada caminho

Usaremos uma matriz para representar o grafo. Ao invés de indicar 0 e 1 para dizer se há ou não caminho entre os itens, usaremos -1 para indicar que não há caminho, e um valor maior que zero para indicar a distância (ou o tempo de percurso) entre dois itens.

Esse grafo pode ser representado pela matriz abaixo, na figura 8. As posições em branco da matriz representam as bordas em que não há saída direta. Não foi colocado o valor -1 para não dificultar a visualização.

	0	1	2	3	4	5	6	7	8	9	10
0											
1			100	507			230		180		
2	110		204								
3		120		97							
4		302				92					
5	88	150									
6				105	113						50
7											
8											
9											
10								74	485		

Figura 9 – Matriz de adjacências como uma estrutura de dados para representar as informações do grafo

Nosso objetivo é desenvolver métodos para resolver os seguintes problemas:

- Encontrar um caminho entre uma cidade A e uma cidade B;
- Encontrar todos os caminhos entre A e B;
- Achar o menor e o maior dos caminhos entre A e B;
- Encontrar circuitos (sair de uma cidade e voltar a ela).

O primeiro problema é semelhante a encontrar um caminho no labirinto. Parte-se de uma cidade A, indicada pelo número de uma linha, e deve-se chegar a uma cidade B. O caminho pode ser ou não

direto, e pode ou não existir. Além disso, podem existir vários caminhos diferentes, passando por várias cidades diferentes.

Já o segundo problema parte desse pressuposto, de que há vários caminhos diferentes. Deve-se encontrar todos eles. A resolução deste problema é uma extensão da resolução do anterior.

Para encontrar o menor caminho dentre todos, deve-se encontrar cada um deles e verificar o menor de todos. O critério de decisão pode ser baseado em número de cidades visitadas ou então em distância total percorrida. No quarto problema, a cidade de destino é a mesma que a cidade de origem.

Vamos tentar desenvolver o método do primeiro problema, e a partir desse você desenvolverá os demais. Suponhamos que os dados estão armazenados na matriz abaixo:

```
int [ , ] grafo = new int[MaxCidades, MaxCidades];
```

Note que pode haver um caminho direto entre a cidade A e a cidade B. Assim, basta verificarmos se $\text{grafo}[A][B]$ é diferente de -1, e caso isso ocorra, já teremos um caminho. Pode até mesmo não ser o menor de todos eles, mas será um caminho, e atende ao solicitado acima.

No entanto, caso não exista um caminho direto, o que faremos? O correto é testar cada possível saída a partir de A, e verificar se existe uma saída. Pode ser que exista uma saída de A que tenha saída para B. Assim, teríamos apenas um desvio. No entanto, pode ainda ocorrer que essa cidade não tenha saída direta para B, de maneira que deveremos tentar novamente outra saída. Isso vai ocorrer até que se encontre a saída para B, ou então que descubramos que não existe caminho entre A e B.

No labirinto, quando se chegava a uma posição sem saída, retornava-se à posição anterior para testar a próxima direção possível. O mesmo deveremos fazer aqui. Sempre que estivermos em uma determinada cidade (digamos uma cidade A^i), e a primeira saída dela demonstrar que não chega a B (direta ou indiretamente), devemos testar a próxima saída de A^i . Se por acaso **todas** as saídas de A^i se mostrarem inviáveis, devemos retornar para a cidade a partir de onde chegamos a A^i , e testar sua próxima saída.

Assim, notamos que o processo pode ser solucionado com o uso de uma pilha e de backtracking.

Suponha que queremos encontrar um caminho entre a cidade 1 e a cidade 5. Note que não existe ligação direta entre as duas. Na verdade, devemos esquecer as ligações diretas, pois elas são apenas um caso especial da seqüência de ligações arbitrárias entre duas cidades.

Partamos da cidade 1, e verifiquemos a primeira saída possível, pela cidade 2 ($\text{grafo}[1][2] \neq -1$). Faremos a cidade 2 ser nosso novo ponto de partida (o problema agora se reduziu a encontrar caminho de 2 para 5), mas antes disso devemos guardar nosso ponto de partida, ou seja, cidade 1 e saída 2. Fazemos isso para o caso de a saída pela cidade 2 se mostrar inviável, e para podermos retornar à cidade 1 e testar as demais saídas que ela possa ter.

Assim sendo, fazemos com que a cidade 2 seja nosso novo ponto de partida. Testamos suas saídas, como fizemos acima (lembre-se que o processo deve ser repetido a cada nova etapa). No entanto, logo na segunda tentativa retornamos à cidade 1, pois $\text{grafo}[2][1] \neq -1$. Isso acarretará LOOP, pois guardaremos a cidade 2 e saída 1 como origem, e voltaremos à cidade 1, de onde partiremos novamente para a 2, de onde partiremos para a 1, e assim por diante, até que ocorra OVERFLOW na pilha (pois a estrutura onde guardaremos os dados de origem é uma pilha, já que quando um caminho se mostra inviável devemos recuperar os dados do ponto de partida anterior). Portanto, para evitar esse LOOP, devemos evitar de seguir novamente para as cidades por onde já passamos. Como fazer isso? No problema do labirinto usávamos a própria matriz para isso, colocando um caractere diferente de espaço nas posições por onde passávamos.

No entanto, neste problema não se deve fazer isso, pois se colocássemos -1 em $\text{grafo}[1][2]$ estaríamos alterando o valor da distância entre as cidades, que será necessário para calcular a distância total (problema 3). Assim sendo, uma solução é usarmos um vetor de valores lógicos que indique se passamos ou não por uma cidade qualquer. Esse vetor começaria com todas as posições

valendo **false**, e a cada empilhamento usaríamos o número da linha atual (= cidade de partida) para indexar a posição correspondente desse vetor e marcá-la com **true**.

Esse vetor seria declarado como:

```
bool[ ] passouCidade = new bool[MaxCidades];
```

E quando saíssemos da cidade de índice **i**, deveríamos executar o comando:

```
passouCidade[i] = true;
```

Continuando, tínhamos chegado à cidade 2, e tentaríamos a sua primeira saída. Como **passouCidade[1] = true**, 1 não é uma saída **válida**, e portanto testamos a próxima. Grafo[2][2] (diagonal principal) vale -1, logo testamos grafo[2][3], que vale 204, e como **passouCidade[3] é false**, essa é uma saída válida. Marcamos **passouCidade[2]** com **true**, empilhamos nossa posição atual e fazemos a linha atual passar a valer 3, de onde testamos as saídas. Como **passouCidade[1] e passouCidade[2] são true**, verificamos que **grafo[3][4] = 97** e que **passouCidade[4] é false**, de modo que podemos passar da cidade 3 para a 4. Novamente empilhamos os dados de origem e marcamos **passouCidade[3]** com **true**. Até este momento, a pilha e o vetor contêm os dados abaixo (figura 10):

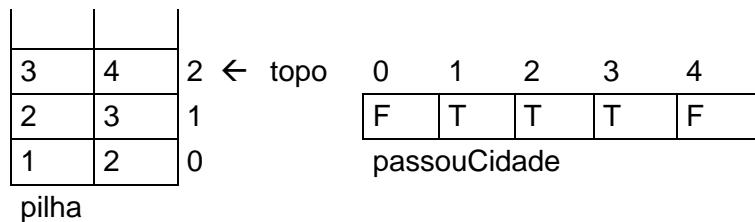


Figura 10 – Representação da pilha e do vetor de passagens por cidades após alguns percursos no grafo

Na cidade 4, verificamos que a primeira saída válida é para a cidade 6. Assim, empilhamos os dados de origem, marcamos a cidade 4, e vamos para a cidade 6. Nessa cidade, descobrimos que a primeira saída válida é pela cidade 5. Empilhamos os dados, marcamos a cidade 6, e passamos para a cidade 5.

Nesse momento, chegamos à cidade desejada. O percurso feito está armazenado na pilha. Para exibi-lo, basta desempilhar os valores da pilha, empilhá-los numa segunda pilha e, em seguida, desempilhá-los todos, e estes sairão na ordem de percurso do caminho encontrado. Podemos, então, parar o processo.

Note que, se você quisesse achar todos os caminhos, deveria retroceder para a cidade anterior e testar a próxima saída, e depois de esgotar todas as possibilidades dessa saída, retornar e testar a saída seguinte, e assim por diante, até terminarem todas as saídas a partir da cidade atual. Quando isso acontecer, deve novamente retroceder e testar as outras saídas, e assim sucessivamente, até retroceder (com os desempilhamentos) para a cidade de origem do problema, e esgotar todas as possíveis saídas dessa primeira cidade.

Vamos desenvolver uma aplicação Windows Forms para implementar e testar esses conceitos e técnicas.

Essa aplicação poderá se chamar, por exemplo, apCidadesBacktracking.

Após criá-la, renomeie o arquivo Form1.cs para FrmCaminhos. Mude também a propriedade (Name) de Form1 para FrmCaminhos.

No formulário, coloque os componentes abaixo:

Controle	Nome	Text	Controle	Nome	Text
Button	btnAbrirArquivo	Abrir Arquivo	Label		Origem
Textbox	txtOrigem		Label		Destino
Textbox	txtDestino		Groupbox		Tipo de grafo
Radiobutton	rbLigacao	Apenas Ligações	Radiobutton	rbDistancia	Com distâncias
Label		Pilha	DataGridView	dgvPilha	
Label		Grafo	DataGridView	dgvGrafo	
Listbox	lsbMovimentos		Button	btnBuscar	Buscar
OpenFileDialog	dlgAbrir				

Ao lado temos um modelo do formulário, com os controles indicados acima.

No Gerenciador de Soluções, adicione a nova classe Movimento, cujo objetivo é armazenar os valores de uma movimentação no grafo, ou seja, a cidade de origem e a cidade de destino de um passo dado no grafo.

Note que esse objeto possui as propriedades Origem e Destino, com acessadores get e set, para armazenar e recuperar os dados dos atributos origem e destino.

Esse objeto é necessário pois nossa pilha é uma pilha de objetos genéricos, e a informação que precisamos armazenar na pilha para uso posterior é o local onde nos encontramos e o local para onde vamos, num passo do percurso.

```
namespace apCidadesBacktracking
{
    class Movimento
    {// onde estou, para onde vou
        private int origem, destino;

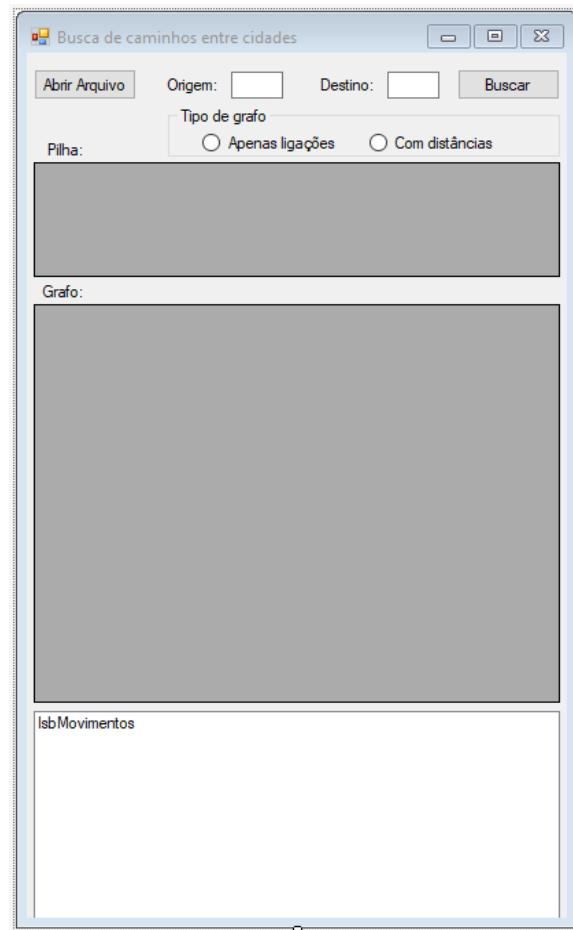
        public Movimento(int or, int dest)
        {
            origem = or;
            destino = dest;
        }

        public int CompareTo(Movimento outro) // compatível com ListaSimples e NoLista
        {
            return 0;
        }

        public int Origem { get => origem; set => origem = value; }

        public int Destino { get => destino; set => destino = value; }

        public override String ToString()
        {
            return origem + " " + destino;
        }
    }
}
```



Abaixo temos a descrição de arquivos texto onde teremos dados de dois grafos.

A primeira linha representa o tipo do grafo. Apenas com ligações (l) ou com as medidas de distância entre as cidades (d).

A segunda linha do arquivo representa a quantidade de cidades do grafo (e, também, o número de linhas do arquivo texto). A partir daí, temos o conteúdo das células do grafo.

O indica que não há saída e um valor maior que 0 indica que há saída da cidade da linha para a cidade da coluna.

No primeiro arquivo, por ligações, o valor 1 indica apenas que existe saída direta de uma cidade para outra, e 0 que não existe.

Já no segundo arquivo, cada valor maior que zero informa a distância entre uma cidade e outra e corresponde ao grafo que usamos na nossa discussão sobre o algoritmo de percurso.

1
10
0 0 1 0 1 0 1 0 0 0
1 0 0 1 0 1 1 0 0 1
0 0 0 1 0 0 0 1 0 1
1 0 1 0 0 0 0 0 1 0
1 1 1 0 0 1 0 0 0 1
0 0 1 0 1 0 0 0 0 1
0 1 1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 1 1 0 1 0 0 0 1 1
1 0 1 0 0 0 1 0 0 0

d
11
0 0 0 0 0 0 0 0 0 0
0 0 100 507 0 0 230 0 180 0
0 110 0 204 0 0 0 0 0 0
0 0 120 0 97 0 0 0 0 0
0 0 302 0 0 0 92 0 0 0
0 88 150 0 0 0 0 0 0 0
0 0 0 0 105 113 0 0 0 50
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 74 485

Vamos, em seguida, criar uma classe GrafoBacktracking, que encapsulará o tipo do grafo, a quantidade de cidades, a matriz de adjacências e vários métodos para buscar o caminho, bem como instruções para mostrar, paulatinamente, o percurso sendo realizado. A classe inicia com o código e os atributos abaixo:

```
using System;
using System.Windows.Forms;
using System.IO;
using System.Threading;

namespace apCidadesBacktracking
{
    class GrafoBacktracking
    {
        const int tamanhoDistancia = 4;
        char tipoGrafo;
        int qtasCidades;
        int[,] matriz;
```

O construtor receberá, como parâmetro, uma string contendo o nome do arquivo com o formato acima descrito, contendo as arestas do grafo. Faremos a abertura e leitura de um arquivo texto, armazenando as ligações na matriz de adjacências:

```
public GrafoBacktracking(string nomeArquivo)
{
    var arquivo = new StreamReader(nomeArquivo);
    tipoGrafo = arquivo.ReadLine()[0]; // acessa primeiro caracter com tipo do grafo
    qtasCidades = int.Parse(arquivo.ReadLine());
    matriz = new int[qtasCidades, qtasCidades];
    for (int linha = 0; linha < qtasCidades; linha++)
    {
```

```
        string arestas = arquivo.ReadLine();
        for (int coluna = 0; coluna < qtasCidades; coluna++)
            matriz[linha, coluna] =
                int.Parse(arestas.Substring(coluna * tamanhoDistancia, tamanhoDistancia));
    }
}
```

As propriedades abaixo permitirão à aplicação acessar os atributos e, também, ajustá-los, quando precisar:

```
public char TipoGrafo { get => tipoGrafo; set => tipoGrafo = value; }
public int QtasCidades { get => qtasCidades; set => qtasCidades = value; }
public int[,] Matriz { get => matriz; set => matriz = value; }
```

O método Exibir() mostrará a matriz de adjacências num DataGridView, passado como parâmetro pela aplicação:

```
public void Exibir(DataGridView dgv)
{
    dgv.RowCount = dgv.ColumnCount = qtasCidades;
    for (int coluna = 0; coluna < qtasCidades; coluna++)
    {
        dgv.Columns[coluna].HeaderText = coluna.ToString();
        dgv.Rows[coluna].HeaderCell.Value = coluna.ToString();
        dgv.Columns[coluna].Width = 30;
    }
    for (int linha = 0; linha < qtasCidades; linha++)
        for (int coluna = 0; coluna < qtasCidades; coluna++)
            if (matriz[linha, coluna] != 0)
                dgv[coluna, linha].Value = matriz[linha, coluna];
}
```

Vamos agora retornar para a aplicação, de forma a desenvolver alguns eventos que chamem os métodos acima. Assim teremos uma compreensão melhor do que faremos na busca dos caminhos desejados.

Crie um tratador de evento Click para o btnAbrirArquivo. Nele solicitaremos ao usuário o nome do arquivo com o grafo, instanciaremos um objeto da classe GrafoBacktracking e faremos a exibição da matriz de adjacências no DataGridView destinado a isso, o dgvGrafo. O código desse tratador de evento segue abaixo:

```
private void btnAbrirArquivo_Click(object sender, EventArgs e)
{
    if (dlgAbrir.ShowDialog() == DialogResult.OK)
    {
        oGrafo = new GrafoBacktracking(dlgAbrir.FileName);
        if (oGrafo.TipoGrafo == '1')
            rbLigacao.Checked = true;
        else
            rbComDistancia.Checked = true;
        oGrafo.Exibir(dgvGrafo);
    }
}
```

Agora, crie um tratador de evento Click para o btnBuscar. Nesse evento, acessaremos os valores da cidade de origem e da cidade de destino desejados, e chamaremos o método BuscarCaminho da classe GrafoBacktracking. Ainda desenvolveremos esse método, mas ele retornará uma pilha contendo os movimentos necessários para trilhar o primeiro caminho encontrado entre a cidade de origem e a cidade de destino.

Se o retorno for uma pilha vazia, significa que não há caminho entre as cidades. Caso contrário, o caminho será exibido no dgvPilha e, também, no Listbox lsbMovimentos. O código do evento vem abaixo:

```
private void btnBuscar_Click(object sender, EventArgs e)
{
    int origem = int.Parse(txtOrigem.Text);
    int destino = int.Parse(txtDestino.Text);
    var pilhaCaminho = oGrafo.BuscarCaminho(origem, destino, lsbMovimentos, dgvGrafo,
                                                dgvPilha);
    if (pilhaCaminho.EstaVazia)
        MessageBox.Show("Não achou caminho");
    else
    {
        MessageBox.Show("Achou caminho");
        pilhaCaminho.Exibir(dgvPilha);
        lsbMovimentos.Items.Add("");
        lsbMovimentos.Items.Add("Caminho encontrado");
        while (!pilhaCaminho.EstaVazia)
        {
            var mov = pilhaCaminho.Desempilhar();
            lsbMovimentos.Items.Add($"De {mov.Origem} para {mov.Destino}");
        }
    }
}
```

O trecho a seguir contém o método BuscarCaminho, que efetua a busca de um caminho entre duas cidades do grafo representado pela matriz. As cidades de origem e de destino são passados como parâmetro. Além disso, os controles de exibição dos passos tomados são também passados como parâmetro.

```
public PilhaLista<Movimento> BuscarCaminho(int origem, int destino, ListBox lsb,
                                              DataGridView dgvGrafo,
                                              DataGridView dgvPilha)
{
    int cidadeAtual, saidaAtual;
    bool achouCaminho = false,
         naoTemSaida = false;
    bool[] passou = new bool[qtasCidades];
    // inicia os valores de "passou" pois ainda não foi em nenhuma cidade
    for (int indice = 0; indice < qtasCidades; indice++)
        passou[indice] = false;
    lsb.Items.Clear();
    cidadeAtual = origem;
    saidaAtual = 0;
    var pilha = new PilhaLista<Movimento>();
    while (!achouCaminho && !naoTemSaida)
    {
        naoTemSaida = (cidadeAtual==origem && saidaAtual==qtasCidades && pilha.EstaVazia);
        if (!naoTemSaida)
        {
            while ((saidaAtual < qtasCidades) && !achouCaminho)
            {
                // se não há saída pela cidade testada, verifica a próxima
                if (matriz[cidadeAtual,saidaAtual] == 0)
                    saidaAtual++;
                else
                    // se já passou pela cidade testada, vê se a próxima cidade permite saída
                    if (matriz[cidadeAtual,saidaAtual] > 0)
                        achouCaminho = true;
            }
        }
    }
}
```

```
if (passou[saidaAtual])
    saidaAtual++;
else
// se chegou na cidade desejada, empilha o local
// e termina o processo de procura de caminho
    if (saidaAtual == destino)
    {
        Movimento movim = new Movimento(cidadeAtual, saidaAtual);
        pilha.Empilhar(movim);
        achouCaminho = true;

        lsb.Items.Add($"Saiu de {cidadeAtual} para {saidaAtual}");
        ExibirUmPasso();
    }
else
{
    lsb.Items.Add($"Saiu de {cidadeAtual} para {saidaAtual}");
    ExibirUmPasso();

    Movimento movim = new Movimento(cidadeAtual, saidaAtual);
    pilha.Empilhar(movim);
    passou[cidadeAtual] = true;
    cidadeAtual = saidaAtual; // muda para a nova cidade
    saidaAtual = 0;           // reinicia busca de saídas da nova
                             // cidade a partir da primeira cidade
}
}
}
}
/// if ! naoTemSaida

if (!achouCaminho)
    if (!pilha.EstaVazia)
    {
        var movim = pilha.Desempilhar();
        saidaAtual = movim.Destino;
        cidadeAtual = movim.Origem;

        lsb.Items.Add($"Voltando de {saidaAtual} para {cidadeAtual}");
        ExibirUmPasso();

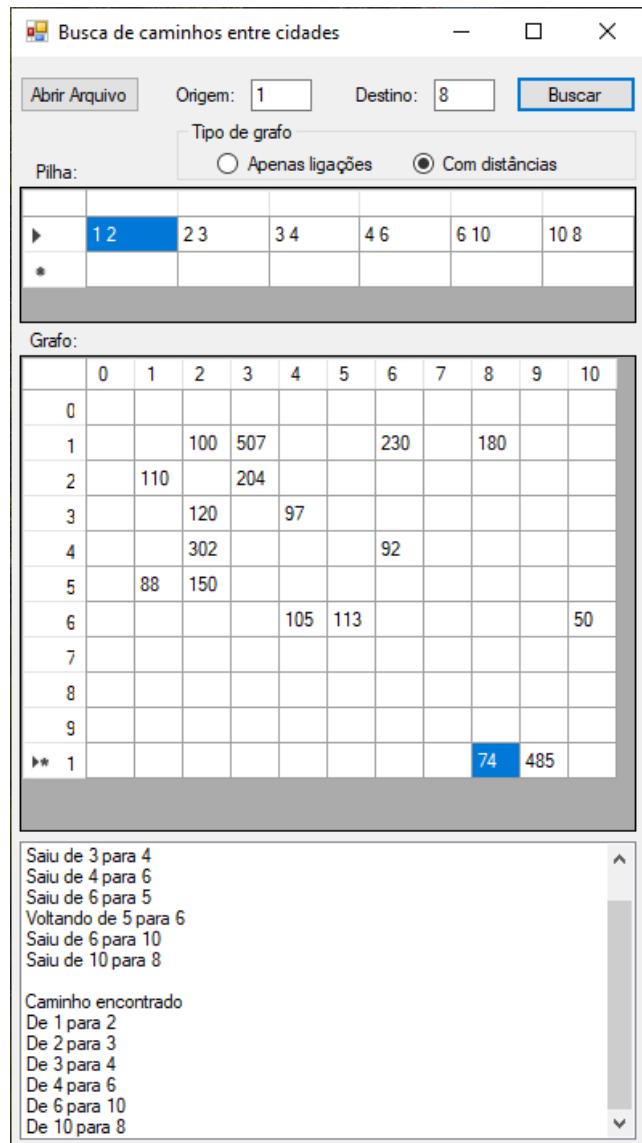
        saidaAtual++;
    }
}

var saida = new PilhaLista<Movimento> ();
if (achouCaminho)
{
    // desempilha a configuração atual da pilha
    // para a pilha da lista de parâmetros
    while (!pilha.EstaVazia)
    {
        Movimento movim = pilha.Desempilhar();
        saida.Empilhar(movim);
    }
}

return saida;

void ExibirUmPasso()
{
    dgvGrafo.CurrentCell = dgvGrafo[saidaAtual, cidadeAtual];
    pilha.Exibir(dgvPilha);
    Thread.Sleep(1000);
}
```

```
}
```



8. Outras Aplicações de Pilhas

Pode-se destacar o controle de processos de sistemas operacionais, chamadas de sub-rotinas, passagem de parâmetros, geração de código executável por compiladores, etc. Normalmente, nessas aplicações são usadas uma pilha interna do programa ou do sistema operacional.

Há também o uso de pilhas para controle de sub-rotinas recursivas, ou seja, que podem ser chamadas de dentro delas mesmas, pois uma chamada recursiva nada mais é do que o empilhamento do endereço de retorno da chamada e empilhamento de parâmetros e variáveis locais.

Em linguagens que não permitem procedimentos recursivos, como FORTRAN e COBOL, processos recursivos devem ser simulados com o uso de pilhas. Os processos acima, de labirinto e grafos, podem ser implementados de maneira recursiva, como teremos oportunidade de estudar.

9. Exercícios

5. Faça um programa em C# que leia, de um arquivo texto, uma matriz de ordem 16 por 21, que representa um labirinto, e terá todos os caminhos de saída para o mesmo. O labirinto tem suas paredes representadas pelo caractere '#'. A matriz propriamente dita terá as linhas 0 e 16, bem como as colunas 0 e 21 preenchidas com esse caractere '#', de modo que você não precisa se preocupar em testar o acesso às bordas da matriz. As direções de percurso são Norte, Leste, Sul e Oeste, de modo que você pode implementá-las com o seguinte vetor constante:

```
int[4] incrementoLinha = {-1, 0,+1, 0};  
int[4] incrementoColuna = { 0,+1, 0,-1};
```

Para mudar de posição, basta somar ao índice linhaAtual o valor incrementoLinha[direcaoAtual]; o mesmo se faz com o índice colunaAtual, ao qual deve ser somado incrementoColuna[direcaoAtual], supondo-se que direcaoAtual define a direção na qual se deseja seguir no labirinto.

Uma saída do labirinto é denotada por um caractere 'S' em alguma das posições da matriz. Ao final, deve-se escrever o caminho encontrado.

6. Em seu programa de pesquisa a grafos dirigidos, como você faria para pesquisar todos os caminhos da cidade A até a cidade B, escrevendo-os e determinando o menor de todos? Supondo que o procedimento ACHAR_CAMINHO está disponível para chamadas, descreva como teria de ser o trecho de programa que o chamassem de modo a encontrar os demais caminhos. Qual terá de ser a condição de parada neste novo processo?
8. Faça um método em C# que receba como parâmetros um tabuleiro de xadrez (Matriz 8x8 de char) e a posição de um cavalo preto no mesmo (linha e coluna), e verifique todas as posições para onde o cavalo pode se dirigir, indicando-as e alertando sobre possíveis peças brancas que possa ameaçar. As peças brancas são representadas por letras maiúsculas e as pretas por minúsculas K, Q, P, B, C, T e k, q, p, b, c e t. Todas as posições possíveis terão sido visitadas quando a pilha estiver vazia e já se tenha testado todos os 8 movimentos possíveis. Lembre-se de que cada posição visitada deve ser marcada para que não ocorra loop, e que quando se esgotar todas as direções, partindo de um ponto, deve-se desempilhar a posição e direção anteriores, para se continuar a partir da próxima direção.

7. Programação Recursiva

1. Definição

Uma subrotina (método, procedimento ou função) é denominado recursivo quando pode ser definido em termos de si próprio. Exemplos:

- i) Fatorial de um número n ($n!$)
se $n = 0 \rightarrow n! = 1$
se $n > 0 \rightarrow n! = n \times (n - 1)!$
- ii) Sequência de Fibonacci
se $n = 0$, ou $n = 1 \rightarrow Fib(n) = n$
se $n > 1 \rightarrow Fib(n) = Fib(n - 2) + Fib(n - 1)$

- iii) Cálculo de expressões aritméticas ou lógicas

Fator ::= Constante | Variável | Função | (Expressão) | not Fator
Termo ::= Fator [Sinal_Mult Fator]...
Sinal_Mult ::= * | / | div | mod | and
Expressão_Simples ::= [+ | -] Termo [Sinal_Adic Termo]...
Sinal_Adic ::= + | - | or
Expressão ::= Expressão_Simples [Sinal_Relac Expressão_Simples]
Sinal_Relac ::= in | < | > | = | <> | <= | >=

A recursão (ou **recorrência, recursividade**) é a técnica que permite desenvolver algoritmos recursivos, geralmente codificados em métodos que **chamam a si mesmos** para resolver um problema.

Internamente, um programa com métodos recursivos utiliza pilhas internas, transparentes ao programador, para gerenciar as diversas chamadas de um método a si mesmo, e também gerenciar a primeira chamada, feita a partir do programa principal ou de algum método auxiliar, chamado originalmente a partir do programa principal. Assim, todo processo que usa pilhas pode ser escrito recursivamente, e vice-versa.

Algorítmos iterativos (repetitivos) como a pesquisa binária, por exemplo, podem ser escritos recursivamente. Num algoritmo recursivo, há uma ou mais chamadas a ele próprio, buscando achar uma solução simples de um problema complexo, através de soluções parciais que se encaminham para a solução final. De certa forma, um algoritmo recursivo é parecido com um algoritmo de Backtracking onde, a partir de um problema mais complexo, efetua-se novas chamadas da função de resolução, cada chamada buscando achar uma solução mais simples que, ao fim do processo, irá compor a solução completa.

No momento em que se chama um método, recursivo ou não, são feitas várias operações transparentes ao programador. O endereço para retorno da chamada é empilhado, bem como os argumentos passados como parâmetros. É por esse motivo que se pode usar recursão para resolver problemas em que uma pilha é a estrutura de dados mais natural. Como exemplo, citamos as árvores binárias e o próprio backtracking, em que vários problemas tem sido propostos, como o percurso do cavalo por todas as posições do tabuleiro de xadrez, como colocar 8 rainhas no mesmo sem que elas se ameacem, Torres de Hanói, Labirinto e grafos dirigidos.

Assim sendo, quando um método é recursivo, ao chamar a si mesmo simplesmente é criada uma nova INSTÂNCIA do método. Os valores das variáveis locais e parâmetros da instância que chama são mantidos intactos, a menos da passagem de parâmetros por referências. Ao entrar na nova instância, as variáveis locais terão valores indefinidos, ao passo que os parâmetros terão os valores passados como argumento na chamada da instância anterior, que não precisam ser os mesmos recebidos quando esta foi chamada. No retorno da nova instância para a instância que a chamou, os antigos valores são restaurados (novamente, os argumentos passados por referência podem ter sido alterados). Dessa maneira, verifica-se que as variáveis locais guardaram os antigos valores, da mesma maneira como se comporta uma pilha.

2. Características de um processo recursivo

1. **Caso Trivial:** o processo deve ter uma **saída bem definida** que, em geral, é a primeira atividade da função que implementa a recursão. Exemplo:

- i) Fatorial : $0! = 1$
- ii) Fibonacci : $\text{Fib}(0) = 0$ e $\text{Fib}(1) = 1$
- iii) Calculo de Expressão : Se há apenas um operando, ele é o resultado (Variável ou Constante)
- iv) Pesquisa Binária: início > fim → não existe a chave procurada
 $\text{vet}[\text{meio}] = \text{chave} \rightarrow \text{achou a chave procurada}$

2. Deve encaminhar-se para uma saída, ou seja, deve chegar a um resultado simples que facilite a resolução das chamadas anteriores. De outra maneira, a rotina poderá entrar em loop.

Exemplos de Algoritmos Iterativos e Recursivos

	Versão Iterativa	Versão Recursiva
FATORIAL	<pre>int n; Console.WriteLine("Digite o valor:"); if (int.TryParse(ReadLine(), out n)) { long f = 1; for (long x=1; x<=n; x++) f = f * x; Console.WriteLine("Fatorial de " + \$"{n} = {f}"); }</pre>	<pre>public long fat(long x) { if (x <= 1) return 1; return x * fat(x - 1); } int n; Console.WriteLine("Digite o valor:"); if (int.TryParse(ReadLine(), n)) Console.WriteLine("Fatorial de " + \$"{n} = {fat(n)}");</pre>
FIBONACCI	<pre>public long fib(int n) { int ant = 0; int prox = 1; for (int i = 2; i<=n; i++) { int x = ant; ant = prox; prox = x+ant; } return prox; }</pre>	<pre>public long fib(long x) { if (x ==0 x == 1) return x; return fib(x-2) + fib(x-1); }</pre>

Quando se resolve um problema com recursão, deve-se verificar em primeiro lugar se existe uma maneira genérica de resolvê-lo. Não se deve procurar uma grande quantidade de possibilidades de resolução; ao contrário, deve-se procurar a solução mais geral. Nos casos acima, de factorial e sequência de Fibonacci, note que a solução geral é retirada exatamente da fórmula matemática recursiva, sem se preocupar com maiores detalhes.

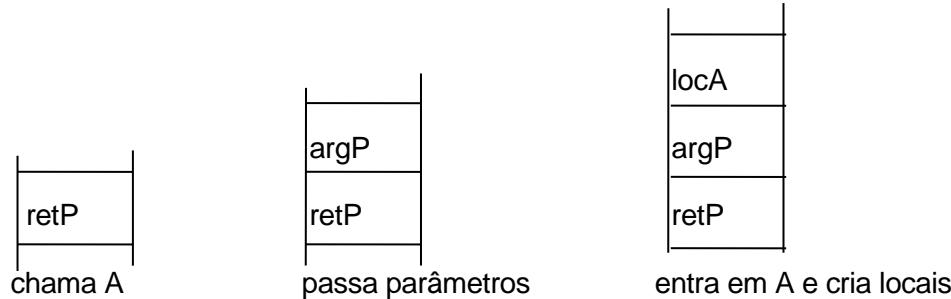
Logo que a solução geral é encontrada, deve-se procurar as condições de parada do algoritmo. No caso de factorial, quando o parâmetro chega a 0, é terminado o algoritmo, de maneira que a função recebe 1. Esse valor da função será transferido para as instâncias anteriores, de maneira que se calculará o produto referente ao factorial do argumento da primeira instância (aquele cujo factorial se deseja calcular). No caso da pesquisa binária, existem duas possibilidades de saída: a primeira quando o índice de início do vetor é maior que o índice de fim, indicando que o elemento procurado não existe (ou o vetor está vazio) e a segunda quando se encontra o elemento. Caso não se verifique nenhuma das duas condições, o algoritmo decide para qual parte do vetor dirigirá a pesquisa.

Ao procurar-se as condições de parada, deve-se ter certeza de que elas serão alcançadas, durante o processamento das diversas instâncias. Se isso não ocorrer, o programa entrará em loop, e diversas instâncias serão criadas em sequência, até que se esgote a memória disponível para a pilha.

3. Funcionamento interno da Recursão

Quando se chama um método, procedimento ou função, é guardado um endereço de retorno para o IP (instruction pointer, ou apontador de instrução do computador), para que se possa voltar ao comando seguinte àquele que efetuou a chamada. Esse endereço é calculado logo no momento da chamada, e é armazenado na pilha do sistema.

No caso de passagem de argumentos para a subrotina chamada, os valores ou endereços dos argumentos (nos casos, respectivamente, de passagem por valor e por referência) são empilhados logo a seguir. Se a subrotina tiver variáveis locais, o espaço para elas é criado no topo da pilha interna do sistema. Assim, para se retornar ao local de chamada, deve-se desempilhar as variáveis locais (que assim deixam de existir após o retorno), os valores e/ou endereços dos argumentos passados e, por fim, o endereço de retorno, que será atribuído ao IP, voltando, assim, o processamento ao comando seguinte ao de chamada da subrotina. A configuração da pilha pode ser representada pela figura abaixo, supondo que P chama A:



`retP` é o endereço de retorno dentro de P

`argP` são os argumentos passados por P para A

`locA` são as variáveis locais de A

Figura 1 – aspecto da pilha interna durante chamada de procedimento com parâmetros e variáveis locais

Se o método chamado efetuar a chamada de um outro método, o mesmo processo acontecerá. O endereço de retorno (agora dentro do método chamado) será empilhado, os argumentos que foram usados nessa nova chamada também (note que variáveis locais ou parâmetros podem ser usados como argumentos nessa nova chamada) e, quando se entrar na nova subrotina, caso ela tenha variáveis locais, estas serão empilhadas. Assim, a configuração da pilha ficaria assim:

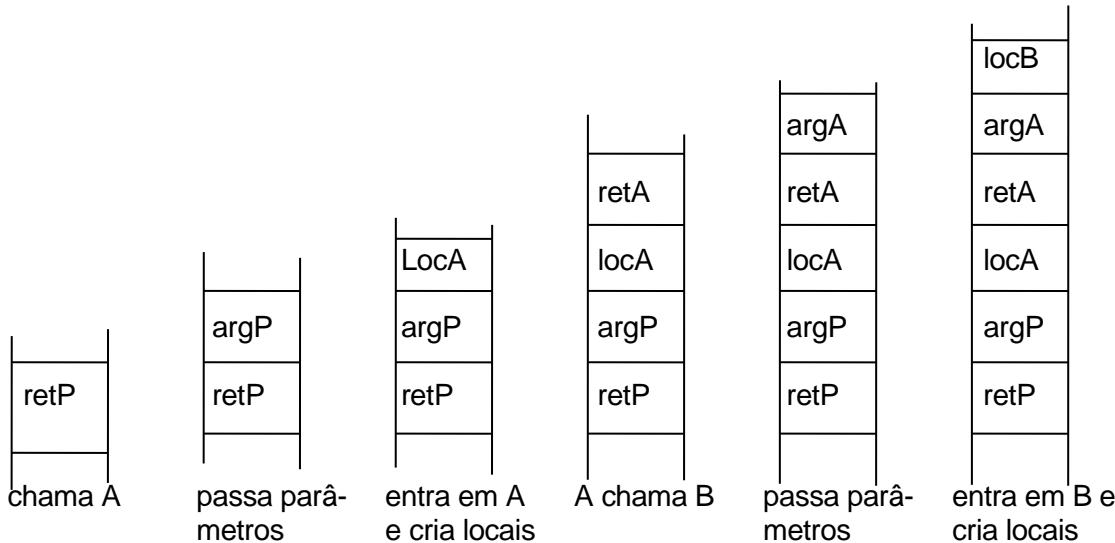


Figura 2 – aspecto da pilha interna com mais chamadas de procedimentos com parâmetros e variáveis locais

Quando se tiver de retornar de B, desempilhar-se-á locB e argA, de forma que o topo da pilha ficará com retA. Esse é o endereço de retorno, que será desempilhado e atribuído ao IP. Assim, o fluxo de execução retorna à subrotina A. A pilha estará da seguinte forma:



Figura 3 – aspecto da pilha interna após retorno de procedimento com parâmetros e variáveis locais

Note que as variáveis locais de B se perderam (viraram “lixo” nas posições seguintes da pilha). As variáveis locais de A continuam com os valores que tinham (a menos que tenham sido usadas como argumentos passados por referência, e que B as tenha modificado). Assim, note que, ao se chamar uma nova subrotina, as variáveis locais na chamada foram mantidas, e depois restauradas. Caso a chamada a B fosse uma chamada RECURSIVA, funcionaria da mesma maneira: as variáveis locais em A seriam preservadas no momento em que se instalasse a nova instância de A, representada por B.

As variáveis locais de B (que seria uma nova instância de A) estariam com valores indefinidos. Os parâmetros recebidos por B viriam da lista de argumentos passados por A (a instância anterior). No caso de passagem por referência, o endereço passado se referiria a uma variável local de A ou mesmo do local de onde A foi chamado (que poderia ser uma instância anterior de A, caso P seja a mesma rotina).

Desenvolvimento de uma solução recursiva de um problema: Pesquisa Binária

PESQUISA BINÁRIA	<pre> public bool Pesquisa(Dado proc, out int onde) { int i = 0; int f = ultimoIndice; bool achou = false; while (i <= f && !achou) { onde = (i + f) / 2; if (vet[onde].CompareTo(proc)==0) achou = true; else if (vet[onde].CompareTo(proc)>0) f = onde - 1; else i = onde + 1; } if (!achou) onde = i; // onde deveria estar return achou; } </pre>	<pre> public bool Pesquisa(int i,int f, out int onde) { if (i > f) { onde = i; // onde deveria estar return false; } else { onde = (i + f) / 2; if (vet[onde] == procurado) return true; if (vet[onde] > procurado) return pesquisa(i,onde-1); return pesquisa (onde+1,f); } } </pre>
------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

No método acima, supomos que **onde** é um atributo do objeto que contém o vetor. Portanto, ao final da execução do método, **onde** contém o índice do elemento procurado dentro do vetor, ou, caso esse valor não esteja armazenado no vetor, **onde** conterá o índice da posição na qual o valor deveria se encontrar, para manter a ordenação dos dados no vetor. Da mesma maneira, procurado seria um membro que conteria o valor a ser procurado no vetor.

4. Diagramas de Execução em Recursão

Uma das maneiras de se verificar o funcionamento correto de um método recursivo é por meio de diagramas de execução, que são quadros que mapeiam os valores das variáveis e a sequência de chamadas de métodos. Como exemplo, estudaremos o método de pesquisa binária. Ele deverá ser chamado do programa principal ou de algum outro método.

Vamos supor que o objeto **Dados**, que encapsula esse método de pesquisa binária possui uma propriedade **Procurado**, que é o objeto que se procura e que foi preenchido pela aplicação. Procurado poderia também ser passado como parâmetro ao método, mas não o fizemos para evitar gastar memória com o empilhamento recursivo dessa variável, que sempre terá o mesmo valor em todas as chamadas.

```
1  public bool Existe(int inicio, int fim, out int onde) // inicio e fim indicam o
trecho
2  {
3      if (inicio > fim) {                                // do vetor em que se pesquisa
4          onde = inicio; // Onde deveria estar
5          return false;
6      }
7
8      onde = (inicio + fim) / 2;                      // calcula a posição média do trecho
9
10     if (vet[onde].CompareTo(Procurado) == 0)           // pesquisado atualmente no vetor
11         return true;
12
13     if (vet[onde].CompareTo(Procurado) > 0)
14         return Existe(inicio, onde-1, out onde);      // Chamada 1
15
16     return Existe(onde+1, fim, out onde);              // Chamada 2
17 }
18 }
```

A primeira chamada de `Existe()` será feita pelo comando abaixo, possivelmente dentro de uma aplicação, na qual a variável inteira **posicao** foi declarada anteriormente::

```
if (Dados.Existe(0, quantosElementos-1, out posicao)) // Chamada 0
```

Mapear o número da chamada é importante, para que não nos percamos durante o diagrama de execução. Por esse motivo, as duas chamadas internas de `Existe()` foram numeradas como chamadas **1** e **2**.

Suponha que a tabela abaixo representa o vetor na memória, e que procuramos o elemento com valor 53.

0	1	2	3	4	5	6	7	8	9	10	11	12
5	6	7	8	12	15	18	48	53	68	75	88	97

Figura 4 – exemplo de vetor ordenado de valores inteiros para pesquisa binária

A lista de argumentos da chamada **0** indica o primeiro índice válido do vetor (temos de procurar desde sua primeira posição), que será associado ao parâmetro **Início**; em seguida, passa-se o índice do último elemento válido (na primeira vez, o elemento procurado poderá estar em qualquer ponto do vetor, logo o intervalo de pesquisa binária vai da primeira à última posições do vetor).

Dessa forma, podemos ver a pesquisa binária como uma verificação do conteúdo de um vetor ordenado dentro do intervalo dado pelos índices Inicio e Fim. Quando se decide que o elemento não está numa das metades desse intervalo de índices, pesquisa-se a outra metade, passando como argumentos o novo intervalo, que irá do valor do parâmetro **inicio** da instância atual até o valor de **onde - 1**, se o elemento procurado for menor do que o encontrado na posição média do intervalo atual de pesquisa, ou de **onde + 1** até o valor do parâmetro **fim** da instância atual, caso o valor procurado seja maior do que o elemento médio do intervalo. Vamos então ao diagrama de execução:

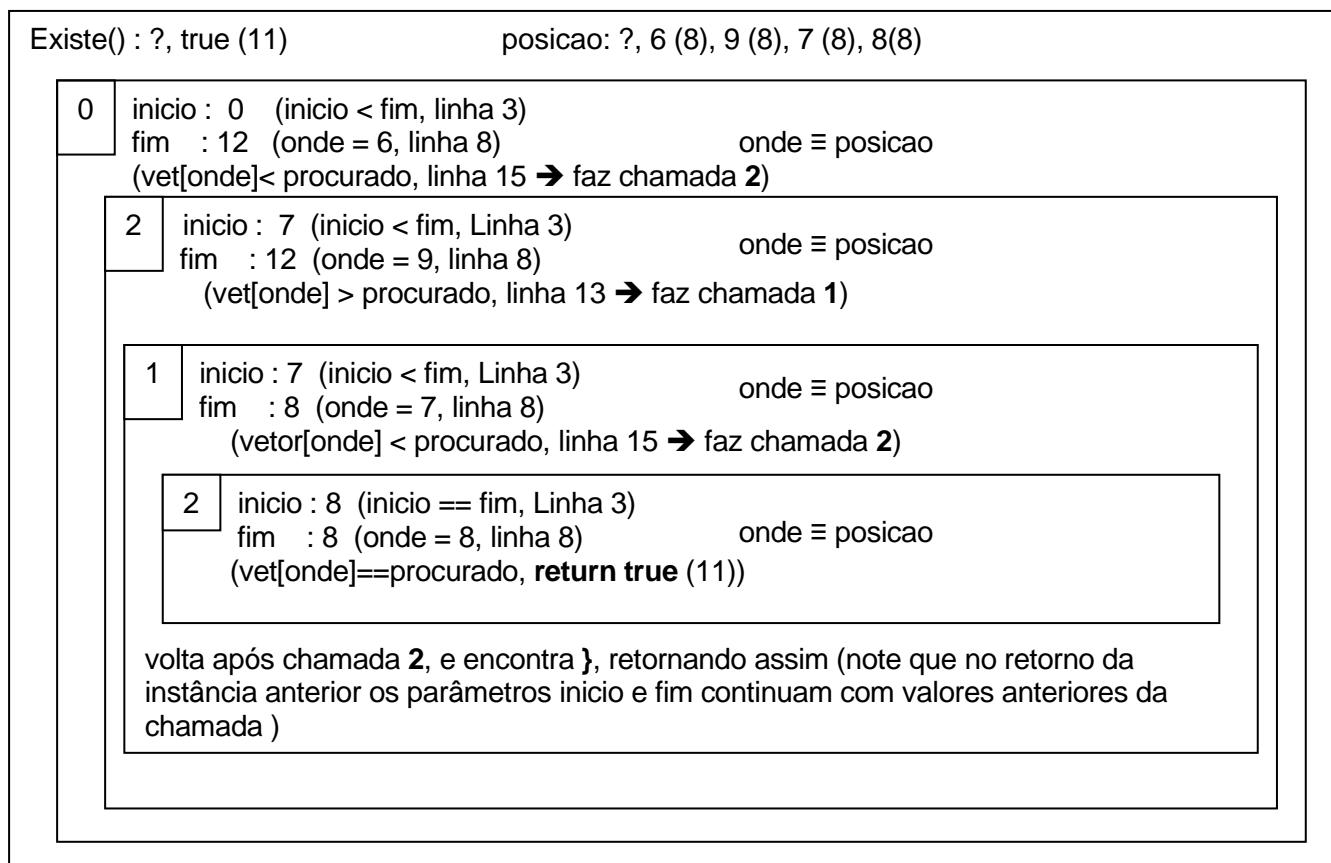


Figura 5 – diagrama de execução da pesquisa binária recursiva do valor 53 no vetor da figura 4

Note que no momento de entrada em uma nova instância do procedimento, os valores dos argumentos são armazenados nos parâmetros. Note que o campo membro **onde** foi recebendo os valores de cálculo da posição média do trecho do vetor que estávamos pesquisando em cada instância.

Abaixo vemos a quantidade de chamadas e repetições efetuadas pelo algoritmo de fatorial na versão recursiva e na versão iterativa, ambas para o $n = 20$. Observe que a versão recursiva é tão ineficiente quanto a iterativa, em termos de repetições/chamadas:

<pre>static long fatR(long n) { quantasSomas++; if (n == 0) return 1; return n*fatR(n - 1); }</pre>	<pre>static double fatI(int n) { double fat = 1; for (long i = 1; i <= n; i++) { quantasSomas++; fat *= i; } return fat; }</pre>
---------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------

Eficiência para $n = 20$ (2.432.902.008.176.640.000):

$\approx O(n)$ 21 somas

$\approx O(n)$ 20 somas

Em seguida, vemos a quantidade de somas efetuadas pelo algoritmo de Fibonacci na versão recursiva e na versão iterativa, ambas para o cálculo do centésimo Fibonacci ($n = 100$). Observe que a versão recursiva é bastante ineficiente em termos de repetições efetuadas:

```
static long fib(long n)
{
    quantasSomas++;
    if (n < 2)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

Eficiência para $n = 45$ (1.134.903.170):

$\approx O(1.59^n)$ 3.672.623.805 somas

$> 10^9$ somas, vários minutos calculando 44 somas, muitíssimo rápido

```
static long fibI(int n)
{
    int ant = 0;
    int prox = 1;
    for (int i = 2; i <= n; i++)
    {
        int x = ant;
        ant = prox;
        prox = x + ant;
        quantasSomas++;
    }
    return prox;
}
```

$\approx O(n)$

Portanto, como decidir quando usar ou não recursão?

Não se deve usar recursão se algoritmos simples iterativos estão disponíveis. Nos casos de Fatorial, Fibonacci e Pesquisa Binária, é muito melhor usar-se as versões iterativas, já que assim não se terá o **overhead** de se chamar procedimentos, com toda a carga de gasto de tempo que isso implica (empilhar endereços de retorno, argumentos, e depois desempilhá-los).

O uso de recursão traz um gasto adicional de memória, pelo próprio uso da pilha interna para armazenamento de endereços de retorno e valores de parâmetros e variáveis locais. Mesmo que as duas versões de Fatorial tenham eficiência semelhante, o tempo gasto em chamadas, passagens de parâmetros e retornos acaba tornando o processo mais demorado.

Já no algoritmo de Fibonacci, por termos chamadas com ramificação em duas subchamadas, temos um crescimento exponencial do tempo de execução.

Na pesquisa binária, embora tenhamos duas chamadas, elas não se ramificam, pois se uma é chamada, a outra não é.

No entanto, algoritmos que utilizam pilhas explícitas podem se beneficiar do uso de recursão, como é o caso dos algoritmos que estudaremos a seguir.

Como exemplos de aplicações em que o uso de recursão não é desaconselhável, podemos citar: backtracking (labirinto, jogos, grafos), percurso em estruturas dinâmicas complexas, como árvores (próximo capítulo), Inteligência Artificial.

5. Aplicações da Recursão

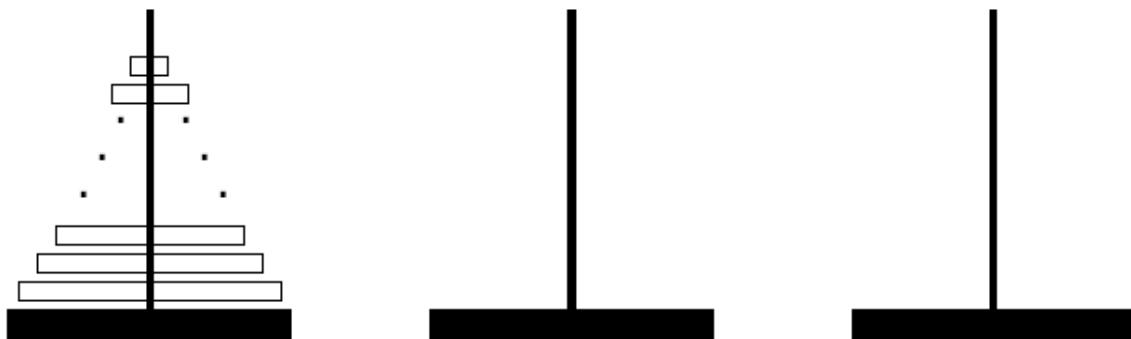
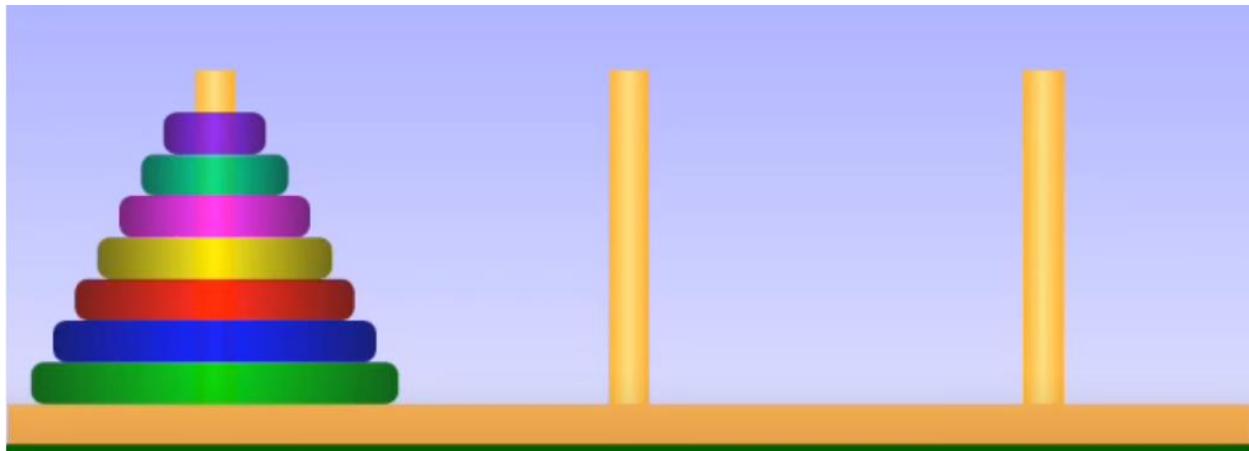
Backtracking

Como vimos em nosso estudo sobre pilhas, uma das tarefas de um computador é resolver problemas através de tentativa-e-erro, ou backtracking. Naquele momento, usamos uma pilha para representar as diversas atividades realizadas para tentar resolver o problema.

Com recursão, podemos dispensar o uso da pilha e usar as diversas chamadas recursivas para armazenar valores intermediários, através de variáveis locais ao método recursivo. Como estudamos anteriormente, as variáveis locais guardam valores que são restaurados no momento do retorno da chamada recursiva, de modo que podemos, dessa maneira, guardar o estado das diversas soluções intermediárias e escolher as que mais se aproximam da solução final.

Torres de Hanoi

[Animação de Torre de Hanoi](#)



A

origem

B

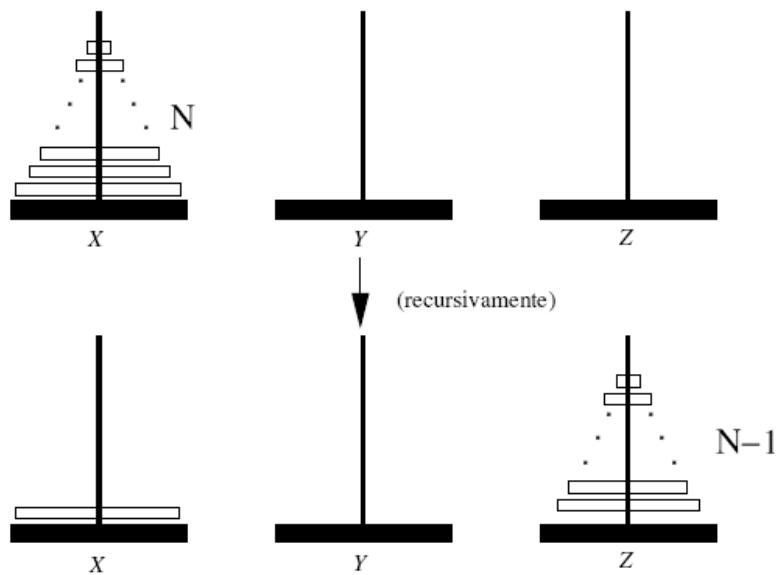
destino

C

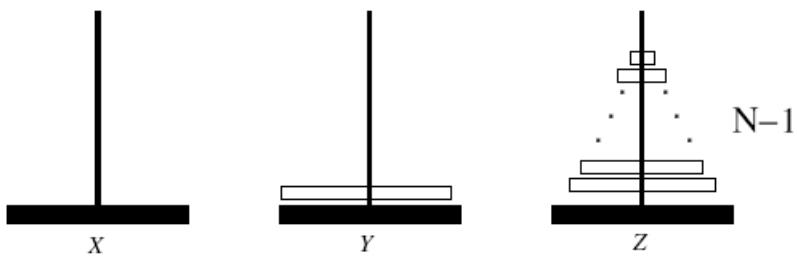
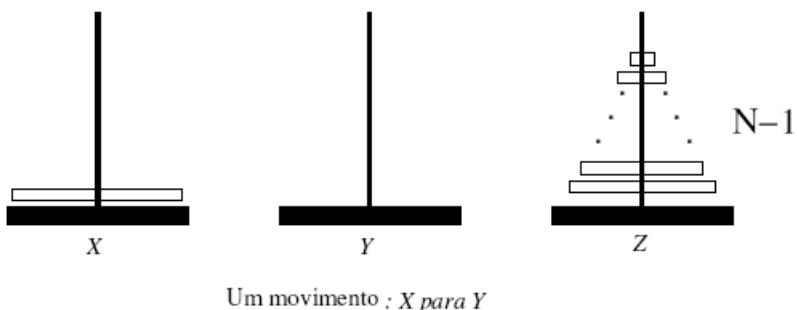
auxiliar

A ideia dessa brincadeira é transferir todos os discos de uma torre para outra delas, mexendo um disco de cada vez e sem que um disco menor fique sobre um disco maior, como vemos nos passos abaixo:

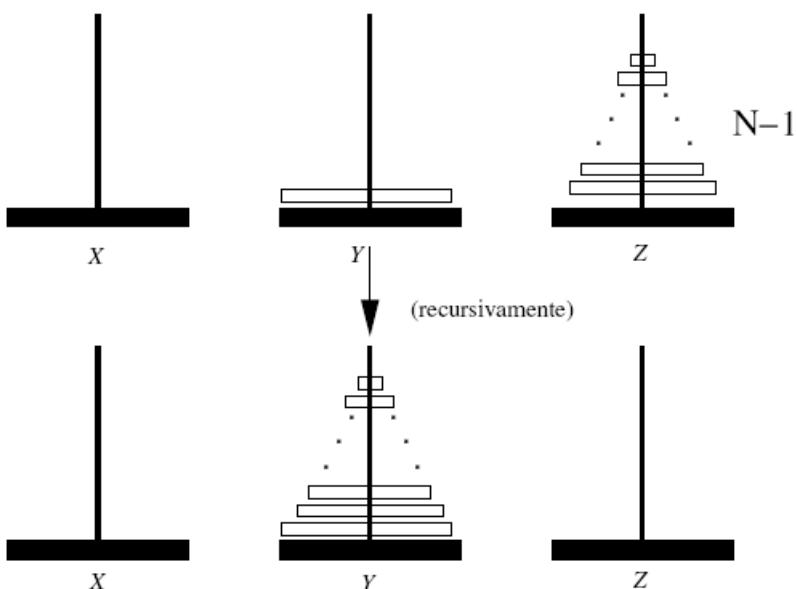
Passo 1: transferência recursiva de $N - 1$ discos, deixando o maior disco livre:



Passo 2: movimento do maior disco



Passo 3: transferência recursiva final de $N - 1$ discos:



A ordem de execução é exponencial: $2^N - 1$ movimentos!

O programa vem abaixo:

```
class Program
{
    static void Hanoi(char org, char dest, char aux, int n)
    {
        if (n > 0)
        {
            Hanoi(org, aux, dest, n - 1);
            Console.WriteLine($"Mova de {org} para {dest}      ");
            Hanoi(aux, dest, org, n - 1);
        }
    }
    static void Main(string[] args)
    {
        Hanoi('X', 'Y', 'Z', 6);
    }
}
```

A saída é a seguinte, para uma torre com 6 discos:

Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de X para Z	Mova de Y para X	Mova de Y para Z
Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de Z para X	Mova de Y para X	Mova de Z para Y
Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de X para Z	Mova de Y para X	Mova de Y para Z
Mova de X para Z	Mova de Y para X	Mova de Z para Y
Mova de Z para X	Mova de Y para X	Mova de Y para Z
Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de X para Z	Mova de Y para X	Mova de Y para Z
Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de Z para X	Mova de Y para X	Mova de Y para Z
Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de Z para X	Mova de Y para X	Mova de Y para Z
Mova de X para Z	Mova de Y para X	Mova de Z para Y
Mova de Z para X	Mova de Y para X	Mova de Z para Y
Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de X para Z	Mova de Y para X	Mova de Z para Y
Mova de Z para X	Mova de Y para X	Mova de Z para Y
Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de Z para X	Mova de Y para X	Mova de Z para Y
Mova de X para Z	Mova de X para Y	Mova de Z para Y
Mova de Z para X	Mova de Y para X	Mova de Z para Y
Mova de X para Z	Mova de X para Y	Mova de Z para Y

Problema das 8 Rainhas

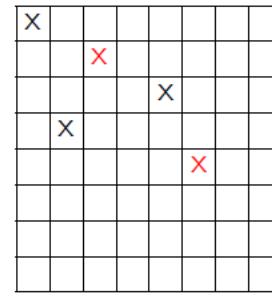
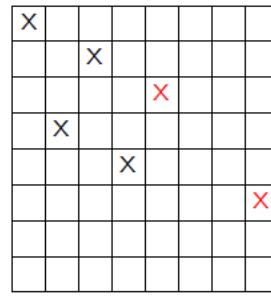
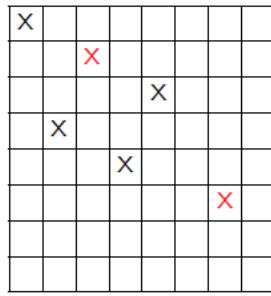
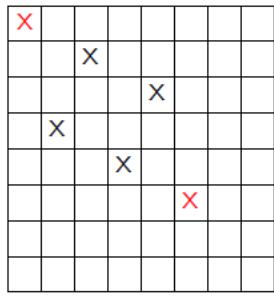
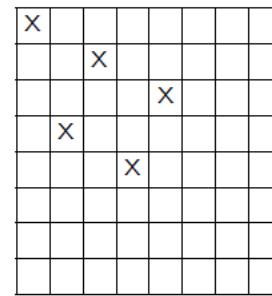
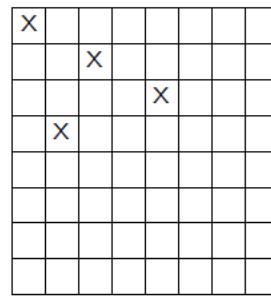
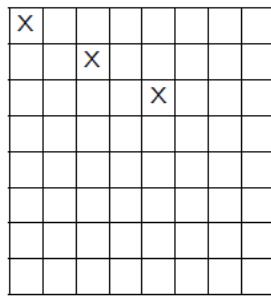
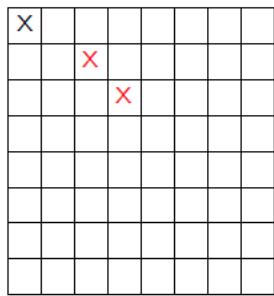
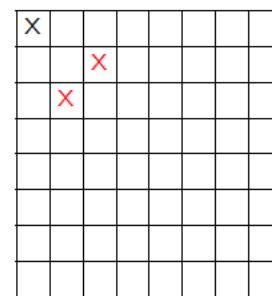
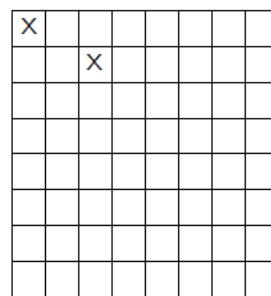
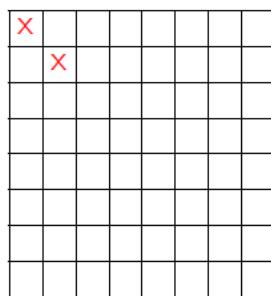
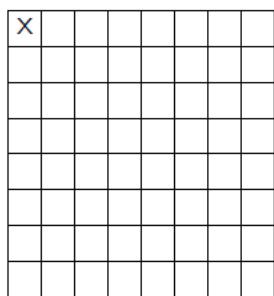
Nesse problema, temos que colocar oito rainhas em um tabuleiro de xadrez de dimensões 8 x 8, sem que nenhuma delas ataque qualquer uma das outras.

Isso implica em que duas rainhas quaisquer não poderão estar dispostas em uma mesma linha, coluna ou diagonal.

Podemos representar uma solução candidata como um vetor **rainhas** de 8 posições, de tal forma que a rainha i seja posicionada na linha i e na coluna $\text{rainhas}[i]$, já que duas rainhas nunca serão posicionadas na mesma linha.

Como duas rainhas também nunca serão posicionadas na mesma coluna, o vetor **rainhas**, quando completo, representará uma permutação dos inteiros de 1 a 8.

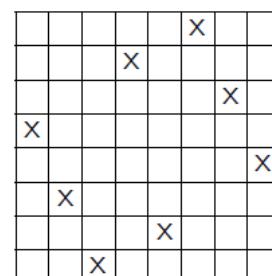
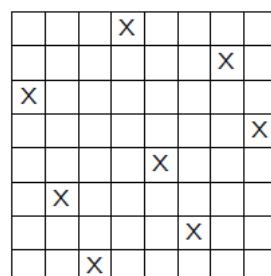
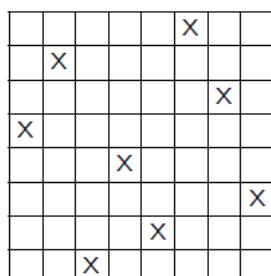
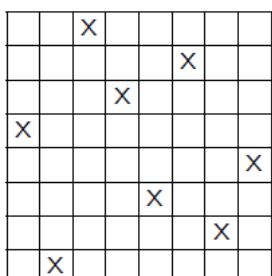
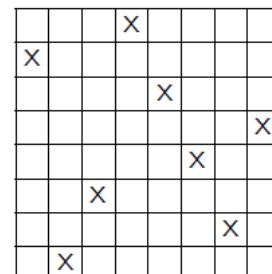
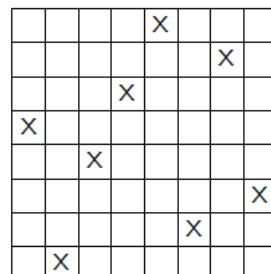
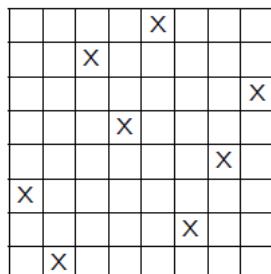
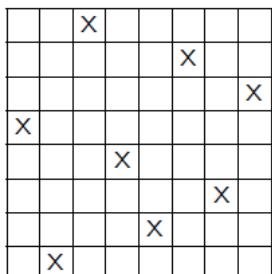
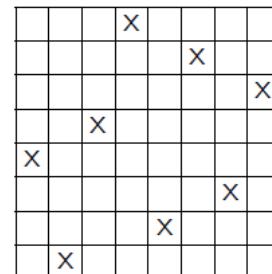
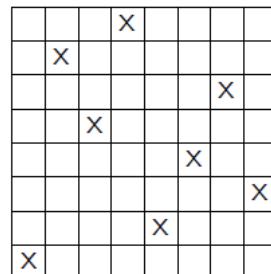
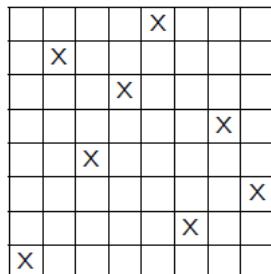
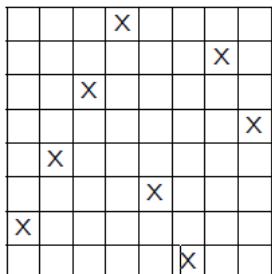
Se algum conflito for detectado em alguma diagonal, o algoritmo dará um passo para trás (backtrack) e tentará reposicionar a última rainha.



O problema das oito rainhas possui 92 soluções distintas, as quais podem ser obtidas a partir de um conjunto de 12 soluções únicas por meio de operações de simetria (reflexão e rotação).

O problema pode ser generalizado de tal forma que o objetivo seja posicionar n rainhas num tabuleiro de dimensões $n \times n$, de tal forma que nenhuma delas seja atacada pela outra.

12 Soluções básicas:



A seguir, temos o código e um exemplo de saída. Baseado nesse código e no exemplo de saída, faça uma aplicação Windows Forms em C# que mostre os resultados em um DataGridView.

```
class Program
{
    static bool Valida(int k, int[] rainhas)
    {
        int i;
        for (i = 0; i < k; i++)
            if ((rainhas[i] == rainhas[k]) ||           // se duas rainhas na mesma coluna ...
                (Math.Abs(rainhas[i] - rainhas[k]) == (k - i))) // ... ou na mesma diagonal
                return false;                                     // solucao invalida
        return true;                                         // solucao valida
    }
    static void nRainhas(int k, int n, int[] rainhas)
    {
        if (k == n)                                       // solucao completa
            Imprimir(rainhas, n);
        else
            for (int i = 0; i < n; i++)
            {
                rainhas[k] = i;
                if (Valida(k, rainhas))
                    nRainhas(k + 1, n, rainhas);
            }
    }
    static void Imprimir(int[] rainhas, int n)
```

```
{  
    int i;  
    for (i = 0; i < n; i++)  
        Write($"{rainhas[i] + 1} ");  
    WriteLine();  
}  
static void Main(string[] args)  
{  
    Write("Número de rainhas: ");  
    int n = int.Parse(ReadLine());  
    int[] rainhas = new int[n];  
  
    // inicialmente nenhuma das n rainhas esta posicionada  
    nRainhas(0, n, rainhas);  
    ReadLine();  
}  
}
```

Utilize esse código para criar uma apresentação das soluções em DataGridViews.

Passeio do Cavalo

<https://www.geeksforgeeks.org/the-knights-tour-problem-backtracking-1/>

Este problema tem o seguinte enunciado:

O cavalo é colocado na primeira casa de um tabuleiro de xadrez vazio e, movendo-se de acordo com as regras de xadrez, deve visitar cada casa exatamente uma vez.

A seguir temos um tabuleiro de xadrez com 8x8 casas. Os números nas casas indicam o número do movimento do cavalo:

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Como já vimos, **Backtracking** funciona como uma maneira incremental de resolver problemas. Em geral partimos de um vetor vazio de soluções e, um a um, adicionamos itens. O significado do item varia de problema a problema. No contexto do Problema do Caminho do Cavalo, um item é um movimento do cavalo. Quando adicionamos um item, devemos verificar se a adição do item atual viola alguma regra/restrução do problema e, se isso acontece, devemos remover o item e tentar outras alternativas.

Se nenhuma das alternativas dá certo, então voltamos para o estágio anterior e removemos o item adicionado nesse estágio anterior. Se retornarmos ao estágio inicial, então dizemos que nenhuma solução existe.. Se o item adicionado não viola regras/restricções, então recursivamente adicionamos itens um a um. Se o vetor de solução se tornar completo, então apresentamos a solução encontrada.

A figura abaixo, à direita, mostra os possíveis movimentos de um cavalo, segundo as regras do xadrez (movimento em L). A figura da esquerda apresenta possíveis movimentos de [0,0] até [4,4]:

	2		1	
3				0
		●		
4				7
	5		6	

0	1	2	3	4
1	4	9	12	
2	10	13	6	3
3	5	2	11	8
4		7	14	
				15

Abaixo temos um possível algoritmo para resolver esse problema. Não é o melhor algoritmo mas nos permite entender o raciocínio recursivo inherent, que é o que nos interessa mais neste estudo.

Se todas as casas foram visitadas

Apresente a solução

Senão

- a) Adicione um dos próximos movimentos ao vetor de soluções e, recursivamente, verifique se este movimento leva a uma solução (um cavalo pode fazer, no máximo, oito movimentos distintos. Neste passo, escolhemos um desses oito).
- b) Se o movimento escolhido no passo acima não leva a uma solução, então remova-o do vetor de soluções e tente um dos outros movimentos possíveis (um movimento alternativo).
- c) Se nenhum dos movimentos alternativos funcionar, então retorne falso (isso removerá o item previamente adicionado na recursão e, se falso é retornado pela chamada inicial, então não há solução).

A seguir temos uma implementação para o Problema do Passeio do Cavalo. Ela exibe uma das possíveis soluções em forma de matriz bidimensional. Os valores de 0 a 63 representam cada movimento do cavalo, na ordem em que resolve o problema.

```
using System;

namespace apPasseioDoCavalo
{
    // Um programa C# para o problema do Passeio do Cavalo
    class PasseioDoCavalo
    {
        static int N = 8;
```

```
// Uma função utilitária que verifica se i,j são índices
// válidos para um tabuleiro de xadrez de tamanho NxN
static bool EhSeguro(int x, int y, int[,] sol)
{
    return (x >= 0 && x < N && // não ultrapassa os limites
            y >= 0 && y < N &&
            sol[x, y] == -1); // não foi visitado
}

// Função utilitária que exibe a matriz de solução na tela
static void ExibirSolucao(int[,] sol)
{
    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < N; y++)
            Console.Write(sol[x, y] + " ");
        Console.WriteLine();
    }
}

// Esta função resolve o problema do Passeio do Cavalo
// usando Backtracking.
// Ela usa, principalmente, a função ResolverPasseioRec() para resolver
// o problema. Ela retorna false se nenhum caminho completo é possível,
// e retorna true quando encontra um caminho, após exibi-lo.
// Observe que pode existir mais que uma solução, mas esta função
// busca e apresenta apenas uma
static bool ResolverPasseioDoCavalo()
{
    int[,] sol = new int[8, 8];

    // inicialização da matriz de solução
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            sol[x, y] = -1;

    // xMove[] e yMove[] definem o próximo movimento do Cavalo
    // xMove[] é para o próximo valor da coordenada x
    // yMove[] é para o próximo valor da coordenada y

    int[] xMove = {2, 1, -1, -2, -2, -1, 1, 2};
    int[] yMove = {1, 2, 2, 1, -1, -2, -2, -1};

    // Supomos que o Cavalo estará, inicialmente, na primeira casa
    sol[0, 0] = 0; // o valor 0 armazenado indica o número do movimento

    // Inicie a partir de 0,0 e explore todos os passeios usando
    // ResolverPasseioUtil()
    if (!ResolverPasseioRec(0, 0, 1, sol, xMove, yMove))
    {
        Console.WriteLine("Não existe uma solução");
        return false;
    }
    else
        ExibirSolucao(sol);

    return true;
}

/* A recursive utility function to solve Knight Tour problem */
static bool ResolverPasseioRec(int x, int y, int movei,
                               int[,] sol, int[] xMove, int[] yMove)
{
```

```
int k, next_x, next_y;
if (movei == N * N)
    return true;

// Tenta todos os próximos movimentos a partir da coordenada atual x, y
for (k = 0; k < 8; k++)
{
    next_x = x + xMove[k];
    next_y = y + yMove[k];
    if (EhSeguro(next_x, next_y, sol))
    {
        sol[next_x, next_y] = movei;
        if (ResolverPasseioRec(next_x, next_y, movei + 1, sol, xMove, yMove))
            return true;
        else
            // backtracking
            sol[next_x, next_y] = -1;
    }
}
return false;
}

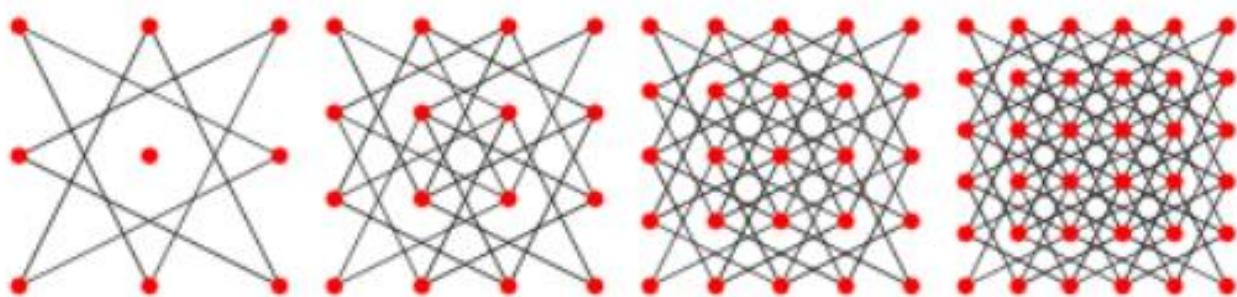
public static void Main()
{
    ResolverPasseioDoCavalo();
    Console.ReadLine();
}
}

// This code is contributed by mits.
```

D:\Chico\0Cotuca\APOSTILAS\EstDad\recu							
0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Na figura acima temos o resultado da execução desse programa.

No link abaixo temos um outro algoritmo, mais eficiente, que deixamos para estudo. Nesse site, o programa desenvolvido apresenta o grafo de movimentos do cavalo no tabuleiro de xadrez (3x3, 4x4, 5x5, 6x6)



Fonte: <https://mathworld.wolfram.com/KnightGraph.html>

Outra referência para estudo:

<https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/>

Outro uso da técnica de recursão é na criação de compiladores, por meio de um conceito chamado compilação descendente. Para isso, usa-se a descrição de uma linguagem por meio de um grafo, chamado **carta sintática**.

Labirinto com solução recursiva

Desenvolveremos o algoritmo que busca um caminho em um labirinto representado por uma matriz. Inicialmente, vamos supor que o arquivo com o labirinto teria as características abaixo :

```
8
10
#####
# # S # #
# ##### # #
# # #     #
# # ##### #
# # #     #
#       # #
#####
#####
```

Figura 6 – exemplo de arquivo texto de definição de um labirinto

Portanto, nosso algoritmo terá de ler os dois valores iniciais, que indicam, respectivamente, o número de linhas e o número de colunas da matriz e, em seguida, as linhas que formam o “desenho” do labirinto, armazenando-as numa matriz de caracteres. Vamos supor, também, que temos 4 direções a seguir na matriz de labirinto, durante o percurso à busca de um caminho de saída (cuja posição é indicada pela letra S, na figura 6).

Portanto, o início da nossa classe LabRec e seu método segue abaixo:

```
class LabRec
{
    int[] LD = new int[4] { -1, 0, 1, 0 };
    int[] CD = new int[4] { 0, 1, 0, -1 };
    char[,] matriz;
    int qtasLinhas, qtasColunas;

    public LabRec(string nomeArquivo) // Construtor
    {
        try
        {
            var arqLab = new StreamReader(nomeArquivo);
            String dados;
            int linha, coluna;

            qtasLinhas = int.Parse(arqLab.ReadLine());
            qtasColunas = int.Parse(arqLab.ReadLine());
            matriz = new char[qtasLinhas, qtasColunas];

            for (linha = 0; linha < qtasLinhas; linha++)
            {
                dados = arqLab.ReadLine();
                for (coluna = 0; coluna < qtasColunas; coluna++)
                    matriz[linha, coluna] = dados[coluna];
            }
            arqLab.Close();
        }
        catch (Exception e)
        {
            WriteLine("Erro no processamento ou abertura do arquivo "+e.Message);
        }
    }
}
```

Após termos lido a matriz, devemos procurar um caminho no labirinto que ela representa. Vamos definir quais serão as direções nas quais podemos nos deslocar:

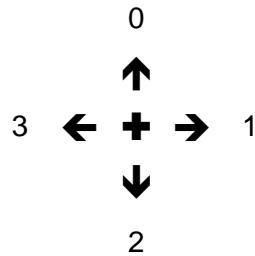


Figura 7 – as 4 direções que usaremos para nos deslocar na matriz

Podemos representar essas direções através de dois vetores de deslocamento, que informam o número de linhas e de colunas que devemos nos deslocar, da posição atual, para chegar à posição dada pela direção desejada.

O nosso ponto de partida é a posição (1,1) da matriz. Assim, nosso problema inicial é sair da posição (1,1) e encontrar um caminho que nos leve à posição da matriz que contém a letra 'S'. Para sairmos dessa posição (1,1), calcularemos uma nova posição, de acordo com o deslocamento previsto para a direção que decidimos tomar. Assim, se a direção atual for a 0, a nova linha será a linha atual menos 1, enquanto a coluna atual é mantida. Dessa maneira, o deslocamento na direção 0 é (-1,0). Nas demais direções temos os seguintes deslocamentos : 1 = (0,1); 2 = (1,0) e 3 = (0, -1). Isso é representado pelos vetores abaixo:

```
int[] LD = new int[4] { -1, 0, 1, 0 };           // deslocamento de linha
int[] CD = new int[4] { 0, 1, 0, -1 };           // deslocamento de coluna
```

Para efetuar o deslocamento, calculamos a próxima linha (PL) e a próxima coluna (PC), a partir da posição atual (L,C) e da direção D :

```
PL = L + LD[D]; // calcula nova posição no caminho
PC = C + CD[D]; // na direção D
```

Observe que, sempre que nos deslocamos de uma posição para outra, o nosso problema passa a ser um novo problema, ou seja, sair da posição (PL,PC) e encontrar um caminho que nos leve à posição da matriz que contém a letra 'S'. Assim sendo, a cada passo que damos na matriz temos um novo problema, e o processo que fazemos para resolvê-lo é o mesmo que fizemos antes. Note também que o ponto de partida de cada novo problema é a posição para a qual nos deslocamos anteriormente, e que, a partir desta nova posição, iniciamos nossa busca por um caminho a partir da direção 0. Ou seja, temos aqui uma recursão como possível solução do processo.

Vamos desenvolver um método HaCaminho() que verificará a existência de um caminho. Ele parte da posição (1,1) e da direção 0, como vemos na sua chamada inicial abaixo, feita no módulo principal do programa:

```
class Program
{
    static void Main(string[] args)
    {
        var labirinto = new LabRec("c:\\temp\\labir.txt");
        labirinto.ExibirMatriz();
        bool achou = false;
        labirinto.HaCaminho(1, 1, 0, ref achou);
        if (achou)
            Console.WriteLine("Encontrou caminho acima \n\n");
        else
```

```
        Console.WriteLine("Não há caminho");
    }
}
```

Abaixo temos o método ExibirMatriz(), que mostrará a matriz na tela console, usando sempre as mesmas coordenadas:

```
public void ExibirMatriz()
{
    SetCursorPosition(0, 0);
    int i, j;
    for (i = 0; i < qtasLinhas; i++)
    {
        for (j = 0; j < qtasColunas; j++)
            Write(matriz[i, j]);
        WriteLine();
    }
    ReadLine();
}
```

O método HaCaminho() recebe como parâmetros a linha, a coluna e a direção atuais, além de uma referência a uma variável lógica que informará se um caminho foi ou não encontrado. Ele deverá, também, calcular a próxima linha e a próxima coluna, partindo da posição atual na direção atual. A declaração desse método na classe LabRec começaria com o código abaixo:

```
public void HaCaminho(int L, int C, int D, ref bool encontrou)
{
    int PL, PC;

    if (D <= 3)
        if (matriz[L, C] == 'S')
    {
        encontrou = true;
        WriteLine("Achou caminho");
        ReadLine();
    }
    else
    {
        PL = L + LD[D]; // calcula nova posição no caminho
        PC = C + CD[D]; // na direção D
        if (matriz[PL, PC] == 'S')
        {
            encontrou = true;
            WriteLine("Achou caminho");
            ReadLine();
        }
    }
}
```

O trecho acima recebe os parâmetros e verifica se não se testou todas as possibilidades de caminho a partir da posição atual (direção $D \leq 3$). Logo após, verifica se a posição atual é a que contém a letra 'S'. Se for, um caminho foi encontrado, e portanto o campo achou recebe **true**. Caso ainda não seja a posição com a letra 'S', calculamos a posição (PL, PC) seguinte e verificamos se esta é a posição final.

Se a posição calculada não for a posição final, devemos verificar se é possível o deslocamento para essa nova posição, ou seja, se ela contiver espaço em branco podemos nos deslocar a ela. Nesse momento, o problema passa a ser sair dessa posição e chegar na posição final; para isso, chamamos, recursivamente, o método haCaminho e passamos como parâmetros a matriz, a posição para a qual nos deslocamos e a direção 0, pois vamos recomeçar a busca a partir dessa posição e para isso temos que reiniciar as direções, também. Vemos isso no trecho a seguir:

```
else
    if (matriz[PL, PC] == ' ')
    {
        WriteLine($"Entrando em Nova Casa {PL}-{PC} direção: {D}");
        matriz[L, C] = 'o';           // marca posição já visitada
        ExibirMatriz();              // para evitar loop

        HaCaminho(PL, PC, 0, ref encontrou);
    }
```

Imagine, agora, que retornamos da chamada recursiva acima e que não encontramos caminho algum. Caso isso seja verdadeiro, devemos tentar a direção seguinte à atual, o que é feito pelo trecho abaixo:

```
if (!encontrou)
    HaCaminho(L, C, D + 1, ref encontrou);
}
```

Imagine, também, que a posição que tínhamos calculado seja uma parede, de modo que não podemos nos deslocar a ela. Assim, devemos também tentar a direção seguinte:

```
else
{
    WriteLine($"Testando nova direção {L}-{C} direção: {D + 1}");
    HaCaminho(L, C, D + 1, ref encontrou);
}
}
```

Perceba que, se nenhuma direção servir, estamos num beco sem saída, e devemos voltar. Poderá acontecer de não existir caminho, de maneira que acabaremos voltando à linha e coluna originais (1,1), com uma direção 4, ou seja, testamos todas as direções possíveis e não encontramos caminho algum, tendo voltado para o início do processo. Nesse caso, temos de parar o algoritmo e não mais procurar o caminho, pois este não existe. Isso é indicado no trecho abaixo:

```
else
    if (L == 1 && C == 1) // voltou ao início sem caminho
        encontrou = false;
```

Após esses comandos, não existem mais chamadas, de maneira que retorna-se para o método main(), com achou valendo false. No entanto, caso encontremos um caminho, podemos aproveitar o retorno das várias chamadas recursivas e exibir o valor da posição atual, que faz parte do caminho, como vemos abaixo:

```
if (encontrou) // achou caminho e o exibe a cada retorno
    WriteLine($"{L} - {C}");
}
```

A classe LabRec completa vem abaixo, já com alguns outros métodos auxiliares para facilitar o acompanhamento do processo na tela:

Note que a exibição do caminho, quando encontrado, será feita ao contrário, ou seja, da última para a primeira posições. Note também que algumas posições poderão sair repetidas. Analise o resultado e dê soluções para as seguintes questões:

1. Como exibir o caminho na ordem correta?
2. Por que aparece mais de uma ocorrência de algumas das posições?
3. Como evitar que essas ocorrências adicionais sejam exibidas?

Um projeto completo de Labirinto recursivo

Tradução de <https://www.c-sharpcorner.com/uploadfile/4a950c/solving-mazes-using-recursion/>.

Nesta lição criaremos um formulário C# que cria e resolve um labirinto através de uma técnica recursiva. A lição cobre a criação do labirinto usando PictureBoxes e a resolução do labirinto. Você também encontrará um arquivo PDF com 9 páginas no download .zip ou este e mais lições em PDF em <http://masteringcsharp.com/>. Espero que goste da minha aula.

Detalhes do Projeto

Objetivos do projeto:

1. Criar um criador de labirinto usando PictureBoxes. Os PictureBoxes são referenciados por uma matriz multidimensional, mazeTiles[,] para fácil acesso e manipulação.
2. Resolver o labirinto usando uma técnica recursiva marcando o caminho correto com a cor cinza. Se o labirinto não puder ser resolvido, mostre uma caixa de diálogo.
3. Faça um botão de reset que removerá a cor cinza e marcará o início e o fim em azul claro. O início é o canto superior direito e o final é o canto inferior esquerdo.

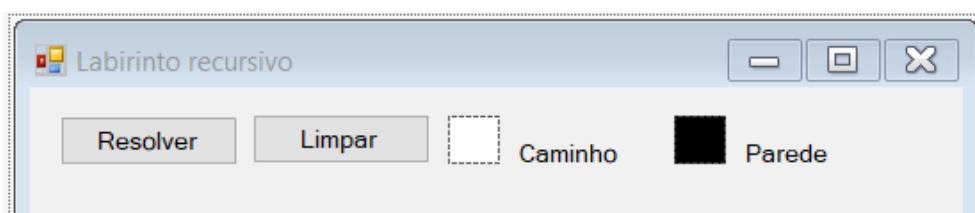


Crie um novo projeto

Estou usando o Visual C # 2010 para esta lição. Crie um novo aplicativo de formulário do Windows. O formulário se chamará FrmLabirinto e seu Text será Labirinto Recursivo.

Criação da interface

Depois de criar seu projeto, adicione ao formulário os controles da figura, usando a caixa de ferramentas no designer de formulário. Observe que os quadrados são PictureBoxes com sua cor de fundo alterada.



Nomeie-os da seguinte forma:

Controle	Nome	Text
Botão	btnResolver	Resolver
Botão	btnLimpar	Limpar
PictureBox	pbBranco	
PictureBox	pbPreto	
Label	label1	Caminho
Label	label2	Parede

Adicionando um Seletor de Cores

O construtor de labirintos permitirá ao usuário criar paredes e caminhos clicando nos ladrilhos do labirinto. A cor será determinada pela última PictureBox selecionada e será o branco por padrão. Crie uma variável global da classe Color nomeando-a corAtual. A adição está abaixo.

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace LabirintoRecursivo
{
    public partial class FrmLabirinto : Form
    {
        Color corAtual = Color.White;
```

Em seguida, torne pbPreto e pbBranco clicáveis adicionando o seguinte tratador de evento a eles, através de um clique duplo no objeto no formulário e digitando o código abaixo em cada método:

```
private void pbBranco_Click(object sender, EventArgs e)
{
    corAtual = Color.White;
}

private void pbPreto_Click(object sender, EventArgs e)
{
    corAtual = Color.Black;
}
```

Criando os blocos do labirinto

O labirinto contém uma grade de PictureBoxes que serão usados para armazenar as paredes, o caminho, o início e o fim, dependendo de sua cor. Vamos tornar o labirinto expansível usando as variáveis XLADRILHOS e YLADRILHOS para determinar o número de peças no labirinto. O tamanho de cada ladrilho será determinado por TAMANHOLADRILHO. Sinta-se à vontade para alterar esses valores para qualquer número não negativo e zero. As PictureBoxes serão armazenadas na matriz **labirinto**. Adicione as seguintes variáveis ao código do seu formulário:

```
public partial class FrmLabirinto : Form
{
    Color corAtual = Color.White;
    private int XLADRILHOS = 25; // Número de ladrilhos em X (horizontal)
    private int YLADRILHOS = 25; // Número de ladrilhos em Y (vertical)
    private int TAMANHOLADRILHO = 10; // tamanho de cada ladrilho (pixels)
    private PictureBox[,] labirinto;
```

Precisamos ter uma maneira de criar o labirinto facilmente com uma chamada de um método. O método CriarNovoLabirinto() criará uma moldura de pictureBoxes. Isso torna o canto superior esquerdo e inferior direito azul claro e todos os outros blocos brancos e clicáveis chamando o método PictureBox_Click (). A última linha, this.Controls.Add (labirinto[i, j]); adiciona os controles PictureBox ao formulário, permitindo que sejam exibidos. Adicione o seguinte código ao seu formulário:

Criação de PictureBoxes do labirinto

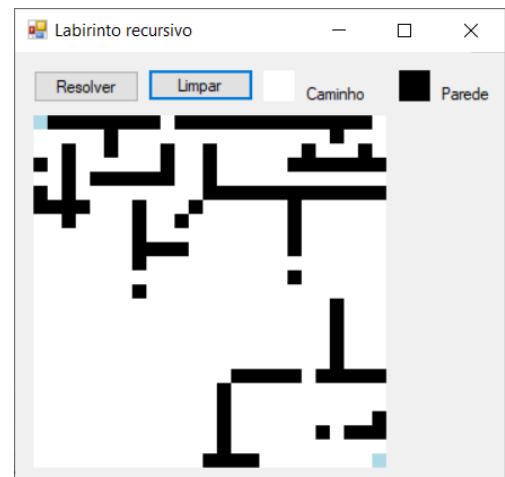
```
private void CriarNovoLabirinto()
```

```
{  
    labirinto = new PictureBox[XLADRILHOS, YLADRILHOS];  
  
    //Loop para obter todos os ladrilhos  
    for (int i = 0; i < XLADRILHOS; i++)  
    {  
        for (int j = 0; j < YLADRILHOS; j++)  
        {  
            // inicializa um novo PictureBox nas coordenadas i, j do labirinto  
            labirinto[i, j] = new PictureBox();  
  
            // calcula tamanho e posição  
            // 13 é preenchimento desde a esquerda  
            int xPosition = (i * TAMANHOLADRILHO) + 13;  
            // 45 é preenchimento desde o topo  
            int yPosition = (j * TAMANHOLADRILHO) + 45;  
            labirinto[i, j].SetBounds(xPosition, yPosition,  
                                      TAMANHOLADRILHO, TAMANHOLADRILHO);  
  
            // Torna azul claro os cantos superior esquerdo e inferior direito.  
            // Indicam início e fim.  
            if ((i == 0 && j == 0) || (i == XLADRILHOS - 1 && j == YLADRILHOS - 1))  
                labirinto[i, j].BackColor = Color.LightBlue;  
            else  
            {  
                // torna todos os demais ladrilhos brancos  
                labirinto[i, j].BackColor = Color.White;  
                // tornar o ladrilho clicável, adicionando um tratador de evento  
                // Click a ele  
                EventHandler eventoClick = new EventHandler(PictureBox_Click);  
                labirinto[i, j].Click += eventoClick; // += usado para o caso de  
                                            // outros eventos serem necessários  
            }  
  
            // Adiciona aos controles do formulario (exibe picturebox)  
            this.Controls.Add(labirinto[i, j]);  
        }  
    }  
}
```

Adicionando o método PictureBox_Click

```
private void PictureBox_Click(object sender, EventArgs e)  
{  
    ((PictureBox)sender).BackColor = corAtual;  
}
```

Execute o seu projeto e você deverá conseguir alterar a cor das PictureBoxes. Você pode experimentar alterar as constantes no topo do arquivo de código para manipular os blocos, mudando sua aresta em pixels e sua quantidade em x e em y.



O Algoritmo Recursivo

A recursão é o ato de um método chamando a si mesmo e é a chave para resolver o labirinto.

Argumentos:

xPos - a coordenada x para pesquisar
yPos - a coordenada y para pesquisar
jáVisitado - matriz bool de elementos pesquisados

Retorna:

bool - é o caminho certo?

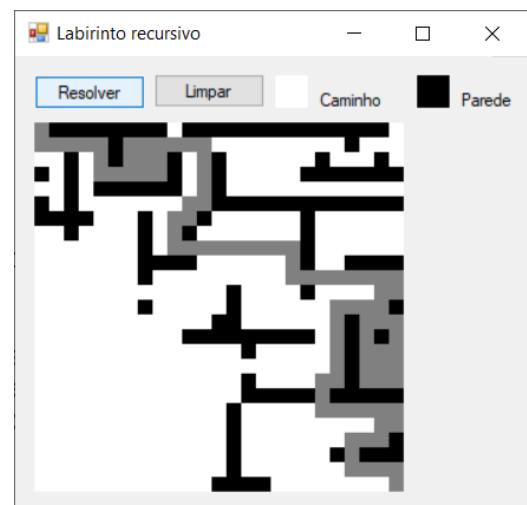
Algoritmo – num determinado ladrilho onde a pesquisa chegou:

se caiu fora do limite, retorna falso
se está em uma parede, retorna falso
se está no final retornar verdadeiro
se não retornado e não jáVisitado, trate o ladrilho

Tratar: Marcando em jáVisitado e verificar caminho à direita, para baixo, à esquerda e para cima e se um deles retornar o caminho correto, retornar true.

Precedência de direção: verifica à direita, para baixo, para cima e para cima.

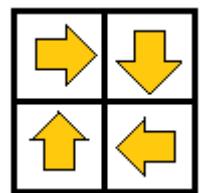
Não será o caminho mais curto.



Tornando o botão Resolver clicável e evitando Loop

Antes de chamar o método recursivo SolucionarLabirinto(), devemos ter uma maneira de evitar um loop infinito. Para evitar isso, adicionamos a matriz bool jáVisitado e a passamos como um argumento para o método recursivo SolucionarLabirinto(). Isso impedirá que caminhos já pesquisados sejam pesquisados novamente.

O método recursivo SolucionarLabirinto() é então chamado, começando no ladrilho 0,0; se retornar falso, uma caixa de mensagem será exibida exibindo "Labirinto não tem solução".



```
private void btnResolver_Click(object sender, EventArgs e)
{
    // Cria uma matriz que informa se a posição foi visitada anteriormente
    bool[,] jaVisitado = new bool[XLADRILHOS, YLADRILHOS];

    // Inicia o solucionador recursivo no ladrilho [0,0].
    // Se retornar falso, o labirinto não pode ser resolvido.
    if(!SolucionarLabirinto(0,0, jaVisitado))
        MessageBox.Show("Labirinto não tem solução.");
}
```

Resolvendo o Labirinto

O código a seguir pesquisará o labirinto um ladrilho por vez e se um caminho correto for encontrado, o caminho será marcado em cinza. Para entender o código, execute-o, bloco por bloco. Você pode diminuir a quantidade de ladrilhos com XLADRILHOS e YLADRILHOS. Adicione o seguinte método ao código de seu formulário:

```
private bool SolucionarLabirinto(int xPos, int yPos, bool[,] jaVisitado)
{
    bool caminhoCorreto = false;

    // o programa deve verificar este ladrilho?
    bool deveVerificar = true;

    // verifica se ultrapassou os limites do labirinto
    if (xPos >= XLADRILHOS || xPos < 0 || yPos >= YLADRILHOS || yPos < 0)
        deveVerificar = false;
    else
    {
        // Verifica se chegou no final, não na posição [0,0] nem cor azul claro
        if (labirinto[xPos, yPos].BackColor == Color.LightBlue &&
            (xPos != 0 && yPos != 0))
        {
            caminhoCorreto = true;
            deveVerificar = false;
        }

        // testa se chegou em parede
        if (labirinto[xPos, yPos].BackColor == Color.Black)
            deveVerificar = false;

        // testa se já visitou essa posição anteriormente
        if (jaVisitado[xPos, yPos])
            deveVerificar = false;
    }

    // Trata o ladrilho
    if (deveVerificar)
    {
        // marca ladrilho como visitado
        jaVisitado[xPos, yPos] = true;

        // Verifica se há caminho indo pelo ladrilho à direita
        caminhoCorreto = caminhoCorreto ||
            SolucionarLabirinto(xPos + 1, yPos, jaVisitado);
        // Verifica se há caminho indo pelo ladrilho abaixo
        caminhoCorreto = caminhoCorreto ||
            SolucionarLabirinto(xPos, yPos + 1, jaVisitado);
        // Verifica se há caminho indo pelo ladrilho à esquerda
        caminhoCorreto = caminhoCorreto ||
            SolucionarLabirinto(xPos - 1, yPos, jaVisitado);
        // Verifica se há caminho indo pelo ladrilho acima
        caminhoCorreto = caminhoCorreto ||
            SolucionarLabirinto(xPos, yPos - 1, jaVisitado);
    }

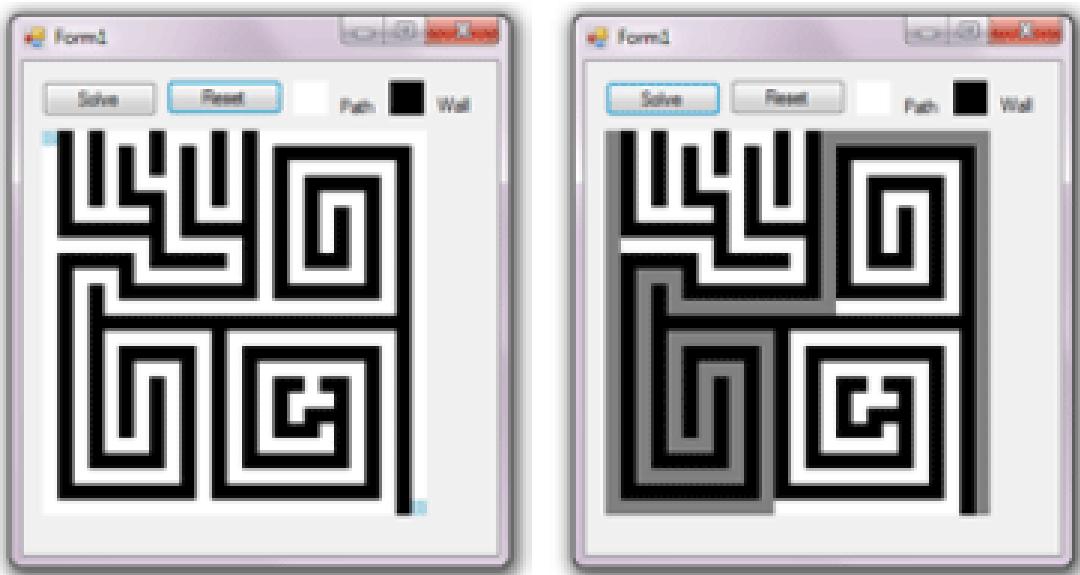
    // torna cinza o caminho correto
    if (caminhoCorreto)
```

```
        labirinto[xPos, yPos].BackColor = Color.Gray;  
  
    return caminhoCorreto;  
}  
}
```

Usando o programa Maze

O solucionador de labirintos agora está completo e você pode começar a criar labirintos para o computador resolver.

Dica rápida: crie um labirinto com várias soluções. Por que o programa escolhe um caminho em vez de outro? Por que não destaca os dois caminhos? Depois de entender as respostas a essas perguntas, você terá um melhor entendimento desse algoritmo e da recursão.



Reiniciando o labirinto

A última coisa necessária é um botão de reinicialização que altera os ladrilhos cinza para branco e o início e o fim de volta para azul claro. Adicione o seguinte código ao Form1.cs e pronto.

```
private void btnLimpar_Click(object sender, EventArgs e)  
{  
    // Altera todos os ladrilhos cinza para branco  
    for (int i = 0; i < XLADRILHOS; i++)  
    {  
        for (int j = 0; j < YLADRILHOS; j++)  
        {  
            if (labirinto[i, j].BackColor == Color.Gray)  
                labirinto[i, j].BackColor = Color.White;  
        }  
    }  
    // Pinta início e fim de azul claro  
    labirinto[0, 0].BackColor = Color.LightBlue;  
    labirinto[XLADRILHOS - 1, YLADRILHOS - 1].BackColor = Color.LightBlue;  
}
```

Eu espero que você tenha gostado deste recurso. Se você tiver dúvidas ou comentários, sinta-se à vontade para me enviar um e-mail para koflaherty@masteringcsharp.com.
Materiais adicionais

Bóson Treinamentos

<https://slideplayer.com.br/amp/1865079/>

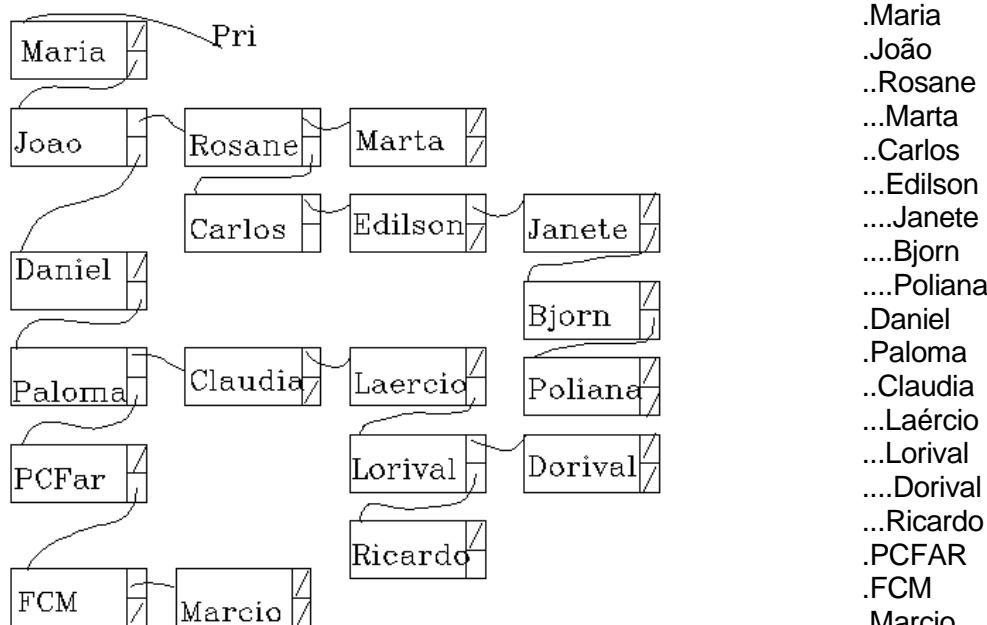
EXERCÍCIO CORRIGIDO

Faça um procedimento em C# que, a partir da lista abaixo, já existente, gere a listagem genealógica correspondente:

Cada nó é composto pelos campos: Nome, Irmão, e Filho. O apontador para baixo é o apontador Irmão, e o para a direita é o apontador Filho.

O procedimento deve percorrer a lista, usando uma pilha para guardar o apontador de retorno para o nó, a cada descendente. Assim, deve-se percorrer, para cada nó, os seus filhos, e após visitar-se todos os filhos, retorna-se e visita-se o irmão do nó desempilhado.

A pilha deve ser implementada com uma lista ligada, sem o uso de vetores. A saída deve ser algo do tipo:



.Maria
.João
..Rosane
...Marta
..Carlos
...Edilson
....Janete
....Bjorn
....Poliana
.Daniel
.Paloma
..Claudia
...Laércio
...Lorival
....Dorival
...Ricardo
.PCFAR
.FCM
.Marcio

A solução com pilhas, portanto, seria:

```
public void Percorrer(ListaSimples<Pessoa> Familia)  {
    NoLista<Pessoa> atual = Familia.Primeiro;
    PilhaLista<NoLista<Pessoa>> pilha = new PilhaLista<NoLista<Pessoa>> ();
    do
    {
        for(nivel=0;nivel<=Pilha. Tamanho;nivel++)
            Console.Write(".");
        Console.WriteLine( (atual.Info.Nome));
        if (atual.Prox != null) // Prox aponta o filho
        {
            pilha.Empilhar(atual);
            atual = atual.Prox;
        }
        else
            if (atual.Info.Irmao != null)
                atual = atual.Info.Irmao;
```

```
else
{
    while (!pilha.EstaVazia() &&
           atual.Info.Irmao == null)
        atual = pilha.Desempilhar();
    if (atual != null)
        atual = atual.Info.Irmao;
}
while (!pilha.EstaVazia() && atual!=null);
}
```

A solução com recursão está feita abaixo:

```
public void Percorrer(Familia atual, int qtos)
{
    int nivel;
    if (atual != null)
    {
        for (nivel=0;nivel<=qtos;nivel++)
            Console.Write(".");
        Console.WriteLine(atual.Info.Nome);
        Percorrer(atual.Prox,qtos+1);
        Percorrer(atual.Info.Irmao,qtos);
    }
}
```

Como exercício, faça o diagrama de execução das duas soluções, comparando-os. Note que o percurso com pilhas é realizado por um algoritmo muito mais complexo que o percurso recursivo. No entanto, durante a execução, este é penalizado por duas chamadas de procedimentos, que são inevitáveis (no percurso iterativo, é possível escrever as operações Empilha e Desempilha diretamente no procedimento de percurso, sem causar o overhead das chamadas).

Este exercício já poderá ter sido resolvido anteriormente por você usando pilhas. O uso de recursão permite a comparação entre os dois métodos.

Mesmo assim, certas operações que envolvem árvores (mesmo disfarçadas, como essas acima), são muito mais facilmente implementadas usando recursão, porque certas estruturas de dados são, desde sua concepção e estruturação, recursivas. É o caso das árvores binárias, que estudaremos em capítulo posterior.

8. Árvores Binárias

1. Introdução

Árvores são estruturas de dados definidas de uma forma hierárquica, ou seja, existe uma relação de hierarquia entre cada um dos elementos da estrutura. Além disso, uma árvore é definida em termos recursivos, ou seja, pode-se usar uma árvore para construir uma árvore.

Assim sendo, definimos uma árvore como um grupo de elementos de dados organizados de tal forma que:

1. Existe um elemento principal chamado de raiz
2. Os demais elementos são subordinados à raiz e formam outras árvores.

2. Definições

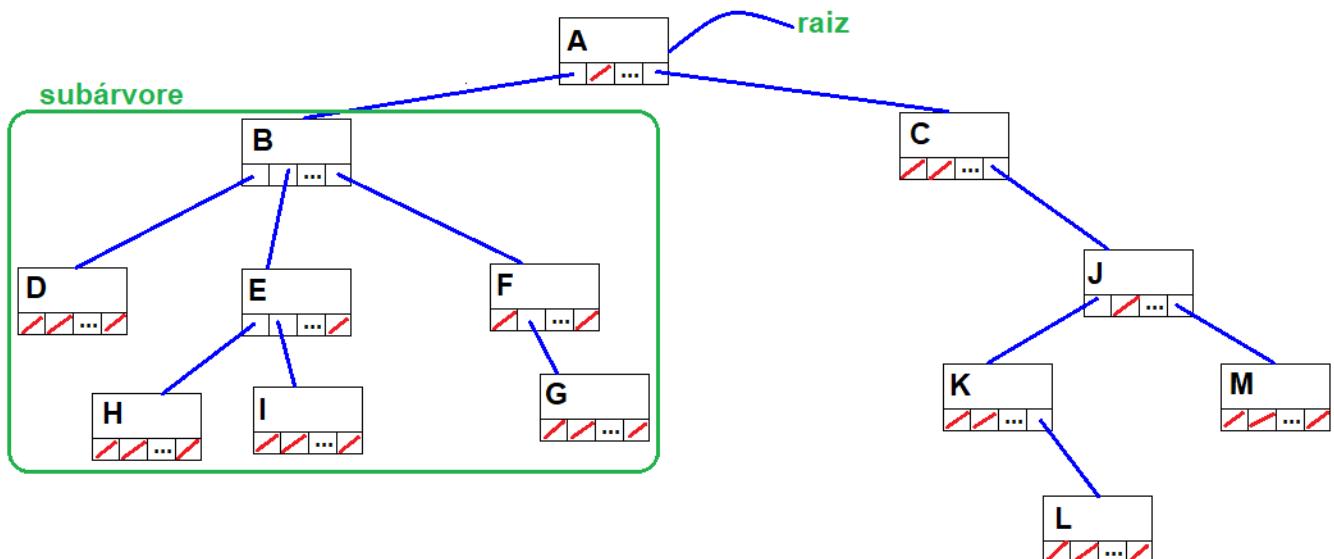


Figura 1 – Uma árvore e seus componentes

Na figura acima, observe que temos vários elementos (que também conhecemos por nós, como vimos em Listas Ligadas) conectados entre si, formando uma relação hierárquica, onde um elemento (que chamaremos de Ancestral) pode dar origem a diversos outros (que chamaremos de Descendentes).

A partir dos elementos apresentados na figura acima, podemos fazer algumas definições sobre esses elementos e como se relacionam:

Raiz: É o elemento da árvore do qual descendem todas as sub-árvores, direta ou indiretamente. É o primeiro elemento da árvore, de certa forma.

Elemento ou Nô: É cada um dos elementos constituintes da árvore, que armazenam informações

Grau: É o número de sub-árvores às quais um elemento dá origem

Folha: Uma folha é um nó de grau zero, ou seja, sem sub-árvores descendentes

Nível: Comprimento do caminho que liga um nó à raiz da árvore

Altura: O maior nível da árvore define a altura da mesma.

Usando essas definições e a figura anterior, a tabela abaixo classifica os elementos da árvore de acordo com seu lugar na hierarquia que a árvore determina:

Nó	Grau	Nível	Observação
A	2	0	raiz
B	3	1	
C	1	1	
D	0	2	folha
E	2	2	
F	1	2	
G	0	3	folha
H	0	3	folha
I	0	3	folha
J	1	2	
K	1	3	
L	0	4	folha
M	0	3	folha

Uma árvore é, portanto, composta por elementos (ou nós), ligados entre si de forma hierárquica. Isso indica que da raiz derivam sub-árvore hierarquicamente inferiores. De cada uma dessas sub-árvore, podem derivar mais sub-árvore, e assim por diante, até que se cheguem às folhas.

Para implementar uma árvore, vamos definir uma classe que armazenará os apontadores principais para acesso aos elementos da árvore e também uma classe que implemente os elementos da árvore:

```
public class NoArvore<Dado>
{
    Dado info;
    NoArvore<Dado> apontador1, apontador2, ..., apontadorN;
    ... outros métodos e campos membros da classe
}

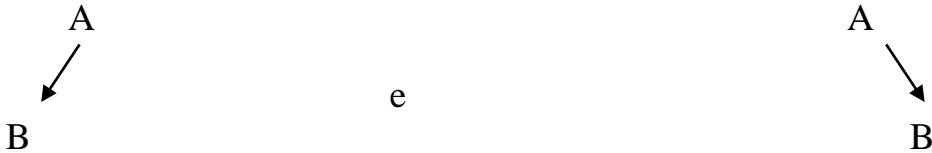
public class Arvore<Dado>
{
    NoArvore<Dado> raiz, atual, anterior;
    public NoArvore<Dado> Raiz { get => raiz; set => raiz = value; }
    ... outros métodos e campos membros da classe
}
```

apontador1, apontador2, ..., apontadorN são apontadores para os nós subordinados ao nó atual. Assim, cada elemento da árvore ou é uma folha (sem nós subordinados), ou é a raiz de uma ou mais sub-árvore (vindo daí a definição recursiva).

3. Árvores Binárias

Uma árvore binária é uma árvore onde cada nó tem **no máximo** grau 2, ou seja, cada nó tem um máximo de duas sub-árvore descendentes. Além disso, as sub-árvore descendentes de um nó qualquer são identificadas como sub-árvore esquerda e sub-árvore direita desse nó. Uma sub-árvore pode ser vazia ou então uma nova árvore binária.

Dessa forma, as árvores abaixo são diferentes, pois suas sub-árvore direita e esquerda não são as mesmas, embora os dados armazenados sejam idênticos:



Em cada nível da árvore binária, o número máximo de nós é 2^i , onde i é o número do nível, de 0 a n (n é maior nível).

Uma estrutura simples da classe de cada nó da árvore binária é a seguinte:

```
class NoArvore<Dado> : IComparable<NoArvore<Dado>> where Dado : IComparable<Dado>
{
    Dado info;
    NoArvore<Dado> esq;
    NoArvore<Dado> dir;
    int altura;

    public NoArvore(Dado Informação)
    {
        this.Info = Informação;
        this.esq = null;
        this.dir = null;
    }

    public NoArvore(Dado dados, NoArvore<Dado> esquierdo, NoArvore<Dado> direito)
    {
        this.Info = dados;
        this.Esq = esquierdo;
        this.Dir = direito;
    }

    public Dado Info { get => info; set => info = value; }
    public NoArvore<Dado> Esq { get => esq; set => esq = value; }
    public NoArvore<Dado> Dir { get => dir; set => dir = value; }

    public int CompareTo(NoArvore<Dado> o)
    {
        return info.CompareTo(o.info);
    }

    public bool Equals(NoArvore<Dado> o)
    {
        return this.info.Equals(o.info);
    }
}
```

O nó do nível 0 da árvore binária deve ser apontado por um apontador especial, que identifica a raiz da árvore. Dessa maneira, continuamos usando a classe Arvore para definir os ponteiros de acesso mais importante para a árvore binária que estamos implementando, além de criarmos métodos que permitirão navegar entre os nós da árvore.

```
Public class Arvore<Dado> where Dado : Icomparable<Dado>
{
    private NoArvore<Dado> raiz, atual, antecessor;
```

```
public NoArvore<Dados> Raiz { get => raiz; set => raiz = value; }
public NoArvore<Dados> Atual { get => atual; set => atual = value; }
public NoArvore<Dados> Antecessor { get => antecessor; set => antecessor = value; }
}
```

raiz apontará o nó raiz da árvore binária, e a partir desse apontador ter-se-á acesso a todos os demais nós. Como se pode notar na próxima figura, a partir de um nó qualquer, pode-se ter acesso a outros dois nós, à esquerda ou à direita. Pode ocorrer também que um determinado nó tenha apenas um descendente direto, ou então nenhum (sendo uma folha).

Sempre que um nó não possuir um nó descendente, o apontador correspondente ao nó descendente deverá conter o valor **null**, indicando que não existe um registro nessa direção.

Para se criar um nó na árvore binária, usa-se alocação dinâmica, da mesma maneira que se fazia com listas ligadas. Note que nesse momento será necessário ligar o nó criado ao seu antecessor, e o nó criado pode ser um descendente à esquerda ou à direita desse antecessor.

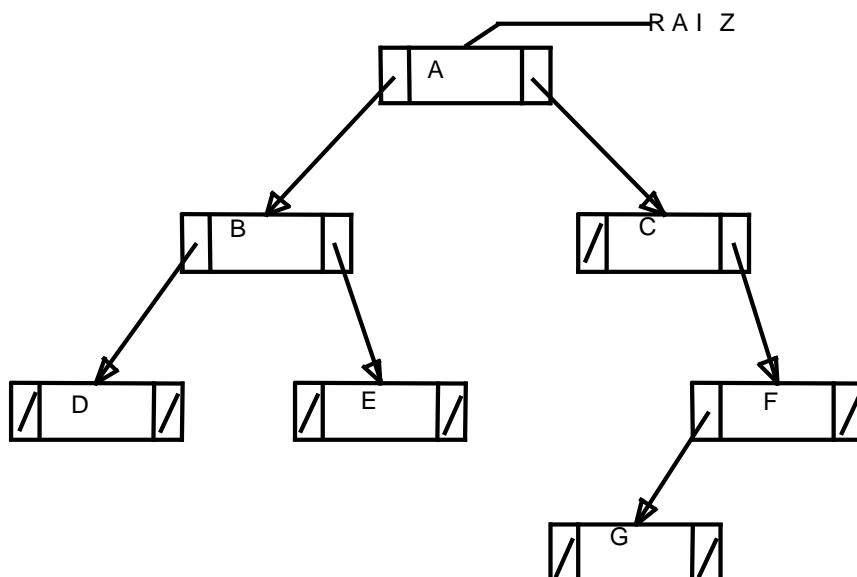


Figura 2 – Árvore binária, com folhas e ponteiros nulos quando não há descendentes em um ramo

4. Percursos em Árvores Binárias

Devido à definição recursiva de uma árvore binária, as operações que podem ser efetuadas nessa estrutura de dados são normalmente bastante fáceis de serem implementadas usando algoritmos recursivos. Quando a recursão não for facilmente implementada, podemos utilizar pilhas para armazenar os apontadores enquanto não se chega ao ponto final do processo. Alguns algoritmos nem mesmo precisam utilizar recursão ou pilhas.

Com a utilização de técnicas de recursão, usa-se a própria definição das operações para se chegar às soluções, de maneira que o resultado reflete o algoritmo de solução de modo bastante aproximado.

As operações usuais que são efetuadas são nossas velhas conhecidas: percurso por todos os elementos, pesquisa de um elemento específico, inclusão e exclusão de elementos específicos. **Percorso** em uma árvore binária é o nome dado ao processo de "visitar" cada nó componente da árvore, uma única vez, de modo que se trate da mesma maneira o nó e suas sub-árvores, ou seja, usando uma abordagem recursiva.

Deve-se lembrar que a árvore não é uma estrutura sequencial como uma lista ligada ou uma fila, de maneira que a abordagem sequencial não é prontamente aplicável. No entanto, para que se visite cada nó da árvore apenas uma vez, temos de controlar o acesso, para que não repitamos acessos.

Como fizemos no desenvolvimento de algoritmos de backtracking recursivos, podemos imaginar as saídas mais simples para visitar a árvore, e então utilizar a recorrência. Assim sendo, qual é a árvore binária mais fácil de se percorrer? É a árvore vazia, em que o apontador da Raiz vale **null**. Portanto, se isso ocorrer, podemos parar o processo de percurso. Em outras palavras, se o apontador Raiz for diferente de **null**, temos que percorrer a árvore. Nesse caso, cabe lembrarmos que estando a Raiz apontando para um nó, podemos já visitá-lo. Essa "visita" pode ser implementada como um acesso aos dados que estão armazenados nesse nó, como por exemplo para escrever alguma informação do nó na tela.

Assim, teríamos algo como:

```
void Visitar(NoArvore<Dados> atual)
{
    if (atual != null)
    {
        Console.WriteLine(atual.Info);
```

Lembre-se que na primeira chamada a esse método Visitar(), o **parâmetro** atual seria associado ao **argumento raiz**. Portanto, na primeira chamada, atual apontará o mesmo nó que raiz.

Após termos visitado esse nó, existem dois caminhos a seguir: pode-se visitar o nó à esquerda ou o nó à direita. Claro que os dois devem ser visitados, já que queremos percorrer **toda** a árvore, mas devemos convencionar uma ordem de percurso. Convencionemos portanto que nossa prioridade será o percurso à esquerda e depois à direita. Esse é apenas uma convenção, mas essa escolha reflete algo que estudaremos na pesquisa em árvores binárias em seguida.

Poderíamos, então, partir para a esquerda da árvore, fazendo **atual** receber o valor de **atual.Esquerdo**. No entanto, o descendente esquerdo da Raiz pode ser raiz de uma sub-árvore (como o nó B da figura 2). Nesse caso, o que fizemos para a Raiz da árvore, devemos fazer para a raiz dessa sub-árvore. Em outras palavras, devemos escrever seu campo **elemento** e, em seguida, passar ao nível seguinte, pois como esse nó é uma raiz de sub-árvore, existem descendentes que devem ser visitados. Isso deve ser feito até que se chegue a uma folha que, por definição, não possui descendentes.

Agora, lembremo-nos que existem também descendentes à direita de cada nó visitado. Assim sendo, depois de visitarmos a descendência esquerda de um nó, devemos visitar a descendência direita. Isso ocorre para cada nó, já que cada descendente pode ser raiz de uma sub-árvore em nível inferior.

Para implementar tal processo, podemos notar que se fizermos uma chamada recursiva ao método VISITA, passando **atual.Esquerdo** como argumento, o parâmetro **atual** na próxima instância apontará para o descendente esquerdo da raiz da sub-árvore antecessora.

```
void Visitar(NoArvore<Dados> atual)
{
    if (atual != null)
    {
        Console.WriteLine(atual.Info);
        Visitar(atual.Esq);1
```

Suponha, por exemplo, que estamos percorrendo a árvore da figura anterior. A Raiz aponta para o nó "A". Na primeira chamada, Atual apontará para esse nó, e println escreverá "A" na tela. Em seguida, é realizada uma nova chamada, em que é passado como argumento o apontador para o nó da esquerda de A. Dessa forma, na nova instância Atual passa a apontar para B. Com Atual é diferente de **null**, "B" é escrito na tela, e nova chamada é realizada, dessa vez passando como argumento o apontador para "D". Mais uma vez, nessa nova instância Atual é diferente de **null**, e assim o println escreve "D" na tela. Em seguida, o apontador para o "filho" esquerdo de D é passado como argumento, em uma nova chamada a VISITA. Na nova instância, Atual valerá **null**, já que D não tem descendente esquerdo. Assim, nada será feito, e retorna-se ao comando seguinte à chamada.

Note que nesse retorno, Atual aponta para o nó D. Percorremos todos os descendentes à esquerda de D (não havia nenhum) e agora temos de percorrer seus descendentes à direita. Para tanto, basta chamar VISITA passando como argumento o apontador para o descendente à direita de Atual. Como se nota pela figura, não há tal descendente, e da mesma maneira, Atual valerá **null** na próxima instância, retornando assim, o fluxo ao comando seguinte a esta chamada.

```
void Visitar(NoArvore<Dados> atual)
{
    if (atual != null)
    {
        Console.WriteLine(atual.Info);
        Visitar(atual.Esq);      1
        Visitar(atual.Dir);     2
    }
}
```

Mas note que já visitamos os descendentes (tanto da esquerda quanto da direita) de D e, portanto, nada mais temos a fazer nesse nível da árvore. Temos que retornar ao nível anterior, para visitarmos a descendência à direita de B (lembre-se: na instância anterior, **atual** aponta para B, e estamos visitando a descendência esquerda de B). Assim sendo, para retornarmos ao ponto de partida não se pode fazer mais nada após a segunda chamada a Visitar(), que ficaria com a seguinte forma final:

```
void Visitar(NoArvore<Dados> atual)
{
    if (atual != null)
    {
        Console.WriteLine(atual.Info);
        Visitar(atual.Esq); 1
        Visitar(atual.Dir); 2
    }
}
```

Quando retornarmos ao nível de B, estaremos retornando após a chamada ¹, e assim, o próximo comando será a chamada ². Essa chamada passa como argumento o apontador à direita de B (Atual = B), que indica o nó E. Na entrada da nova instância, Atual valerá E, sendo diferente de **null**, e assim "E" será escrito na tela, e em seguida os seus descendentes à esquerda e à direita serão visitados. Como E é uma folha, nas duas chamadas Atual valerá **null**, retornando-se então para a instância imediatamente anterior, em que Atual aponta para o nó A. O retorno dar-se-á após a chamada ¹, de maneira que uma chamada para VISITA à descendência direita da Raiz será efetuada. O processo será então repetido. O diagrama de execução desse percurso vem abaixo; os endereços de cada nó foram considerados como sendo a letra referente ao nó respectivo.

Para a árvore que usamos na discussão (a da figura 2), a saída teria o aspecto abaixo:

A B D E C F G

Esse tipo de percurso é chamado de percurso PRÉ-ORDEM, pois a Raiz de cada sub-árvore foi visitada (suas informações foram processadas) antes de qualquer outro elemento.

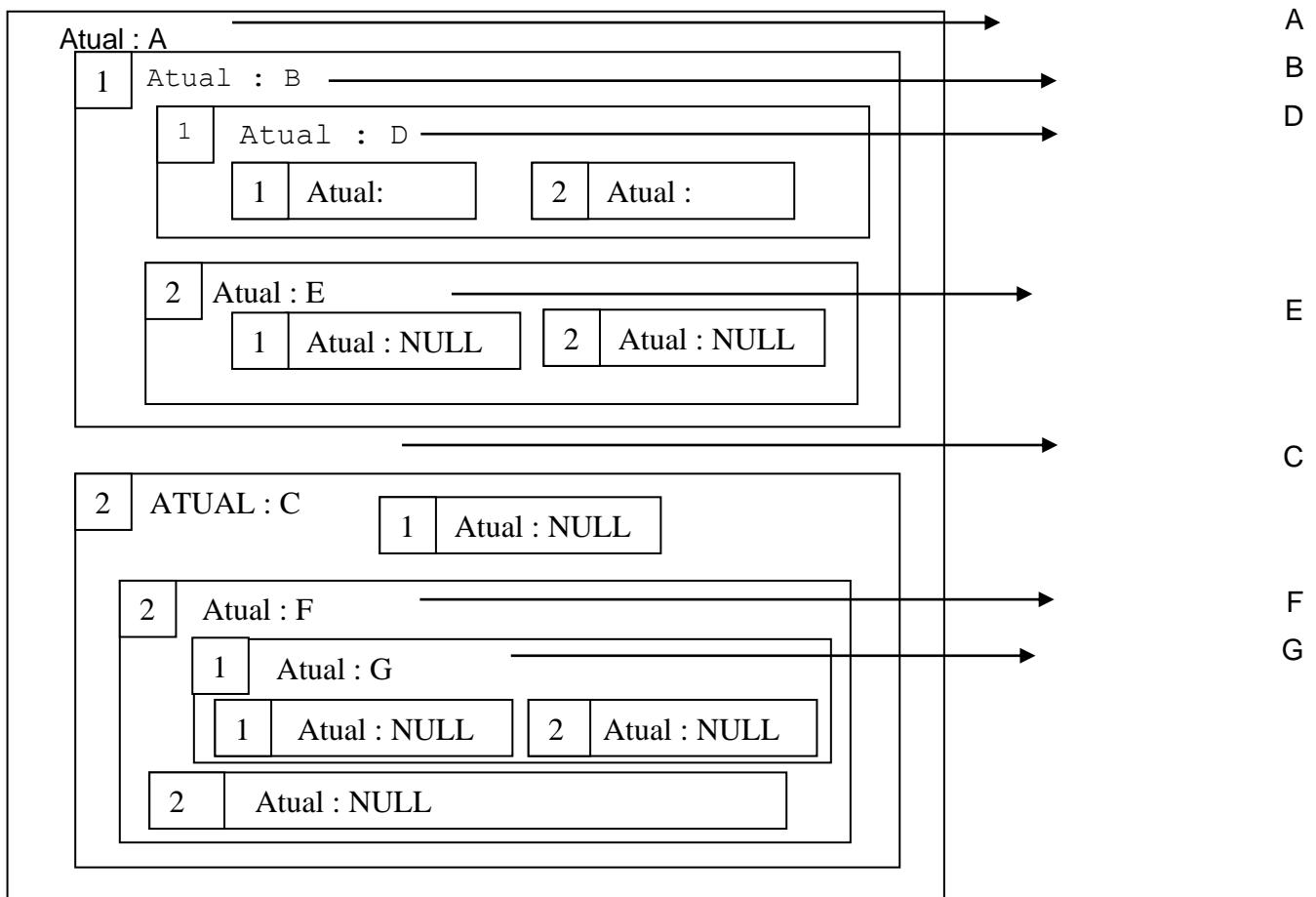


Figura 3 – Diagrama de execução mostrando as chamadas recursivas do percurso pré-ordem de uma árvore binária

Existem ainda os percursos IN-ORDEM, em que a raiz de cada sub-árvore é processada, respectivamente, entre o processamento do ramo esquerdo e do ramo direito, e PÓS-ORDEM, em que o processamento ocorre após ter-se processado os descendentes esquerdo e direito. Resumindo, temos os seguintes percursos:

Pré-Ordem : Raiz, Ramo Esquerdo, Ramo Direito
In-Ordem : Ramo Esquerdo, Raiz, Ramo Direito
Pós-Ordem : Ramo Esquerdo, Ramo Direito, Raiz

O método **Visitar()** que descrevemos acima serviria para percorrer a árvore usando a disciplina do percurso Pré-Ordem. Para executar esse método pela primeira vez, será necessário chamá-lo, possivelmente na aplicação, com o método `VisitarPreOrdem`, que descrevemos abaixo e é membro da classe `Arvore`. Note que esse método é público e não necessita da raiz como parâmetro. Ele é que chama o método `Visitar()` passando a raiz da árvore como parâmetro. `Visitar()` não é público e, por isso, não pode ser chamado pela aplicação. Usando essa estratégia, evitamos que a aplicação tenha de se preocupar em passar a raiz da árvore como parâmetro. Protegemos, de certa maneira, esse ponteiro e tornamos a aplicação mais simples.

```
public void VisitarPreOrdem()
{
    visita(Raiz);
}
```

5. Exercícios

1. Duas árvores binárias são ditas **equivalentes**, ou seja, A **eq** B, se:

- . ambas são vazias
- ou
- .. Info(A) = Info(B) e
- ... Esq(A) **eq** Esq(B) e Dir(A) **eq** Dir(B)

Desenvolva um algoritmo que verifique se duas árvores binárias apontadas pelos parâmetros A e B são equivalentes.

2. Faça uma função recursiva que devolva o número de nós de uma árvore binária.
3. Faça uma função que conte quantas folhas existem em uma árvore binária.
4. Uma árvore é dita ESTRITAMENTE BINÁRIA se todo nó que não é folha possui ambas as sub-árvores não vazias. Faça uma função lógica que verifique se uma árvore, cuja raiz é recebida como parâmetro, atende ou não essa definição.
5. Escreva uma função que calcule a altura de uma árvore binária.
6. Faça um procedimento que escreva a estrutura de uma árvore binária de acordo com a forma abaixo:

(chave : AE , AD)

onde chave é o chave da raiz, AE denota o ramo esquerdo e AD o ramo direito, descendentes da raiz da sub-árvore atual. O procedimento deve escrever apenas a chave quando estiver visitando uma folha. Para a árvore do exemplo, teríamos (A : (B : (D: () , E: ()) , (C : () , (F : (G : ()) , ()))))

7. Escreva um procedimento que troque entre si os ramos esquerdo e direito de uma árvore e de suas sub-árvores.
8. Escreva um procedimento que percorra uma árvore binária por níveis. A raiz é visitada em primeiro lugar, depois os descendentes imediatos da raiz, em seguida os descendentes do 2º nível, e assim por diante. Dica: use uma fila para organizar a seqüência de escrita.
9. Faça uma função que devolva a largura de uma árvore binária, ou seja, o número máximo de nós no mesmo nível.
10. Escreva um procedimento que exiba todos os antecessores de um determinado nó da árvore.

Correção de Exercícios

Exercício 1 – Árvores equivalentes

```
private bool Eq(NoArvore<Dado> atualA, NoArvore<Dado> atualB)
{
    if (atualA == null && atualB == null)
        return true;
    else
        if ((atualA == null) != (atualB == null)) // apenas um dos nós é
            return false; // nulo
        else // os dois nós não são nulos
            if (atualA.Info.CompareTo(atualB.Info) != 0)
                return false; // Infos diferentes
            else
                return eq(atualA.Esq, atualB.Esq) &&
                    eq(atualA.Dir, atualB.Dir);
}
```

```
public bool EquivaleA(Arvore<Dado> outraArvore)
{
    /*
        . ambas são vazias
        ou
        .. Info(A) = Info(B) e
        ... Esq(A) eq Esq(B) e Dir(A) eq Dir(B)
    */
    return Eq(this.raiz, outraArvore.Raiz);
}
```

Exercício 2 – Contagem de nós em árvore, com função recursiva

```
public int QtosNos(NoArvore<Dado> noAtual)
{
    if (noAtual == null)
        return 0;
    else
        return 1 +          // conta o nó atual
                QtosNos(noAtual.Esq) +      // conta nós da subárvore esquerda
                QtosNos(noAtual.Dir);     // conta nós da subárvore direita
}
```

Exercício 3 – Contagem de folhas de árvore, com função recursiva

```
public int QtasFolhas(NoArvore<Dado> noAtual)
{
    if (noAtual == null)
        return 0;

    if (noAtual.Esq == null && noAtual.Dir == null) // noAtual é folha
        return 1;

    // noAtual não é folha, portanto procuramos as folhas de cada ramo e as contamos
    return QtasFolhas(noAtual.Esq) +          // conta folhas da subárvore esquerda
           QtasFolhas(noAtual.Dir);          // conta folhas da subárvore direita
}
```

Exercício 4 – Árvore Estritamente Binária

```
public bool EstrictamenteBinaria(NoArvore<Dado> noAtual)
{
    if (noAtual == null)
        return true;

    // noAtual não é nulo
    if (noAtual.Esq == null && noAtual.Dir == null)
        return true;

    // um dos descendentes é nulo e o outro não é
    if (noAtual.Esq == null && noAtual.Dir != null)
        return false;

    if (noAtual.Esq != null && noAtual.Dir == null)
        return false;

    // se chegamos aqui, nenhum dos descendentes é nulo, dai testamos a
    // "estrita binariedade" das duas subárvores descendentes do nó atual
    return EstrictamenteBinaria(noAtual.Esq) && EstrictamenteBinaria(noAtual.Dir);
}
```

Exercício 5 – Altura de uma árvore binária

```
public int Altura(NoArvore<Dados> noAtual)
{
    int alturaEsquerda,
        alturaDireita;
    if (noAtual == null)
        return 0;

    alturaEsquerda = Altura(noAtual.Esq);
    alturaDireita = Altura(noAtual.Dir);

    if (alturaEsquerda >= alturaDireita)
        return 1 + alturaEsquerda;

    return 1 + alturaDireita;
}
```

Exercício 6 – Escrever a estrutura da árvore no formato (Chave : AE, AD)

```
public string EntreParenteses(NoArvore<Dados> noAtual)
{
    string saida = "(";
    if (noAtual != null)
        saida += noAtual.Info + ":" +
            EntreParenteses(noAtual.Esquerdo) +
            "," +
            EntreParenteses(noAtual.Direito);

    saida += ")";
    return saida;
}
```

Exercício 7 – Trocar ramos esquerdo e direito entre si

```
public void Trocar(NoArvore<Dados> noAtual)
{
    if (noAtual != null)
    {
        NoArvore<Dados> auxiliar = noAtual.Esq;
        noAtual.Esq = noAtual.Dir;
        noAtual.Dir = auxiliar;
        Trocar(noAtual.Esq);
        Trocar(noAtual.Dir);
    }
}
```

Exercício 8 – Percorso por níveis

```
public string PercorsoPorNiveis(NoArvore<Dados> noAtual)
{
    string saida = "";
    FilaLista<NoArvore> umaFila = new FilaLista();
    while (noAtual != null)
    (
        if (noAtual.Esq != null)
            umaFila.enqueue(noAtual.Esq);
        if (noAtual.Dir != null)
            umaFila.enqueue(noAtual.Dir);
```

```
    saida += " "+noAtual.Info;
    if (umaFila.EstaVazia)
        noAtual = null;
    else
        noAtual = umaFila.Retirar();
}
return saida;
}
```

Exercício 9 – Largura de uma árvore binária

```
int[1000] quantosNoNivel = new int[1000]; // GLOBAL NA CLASSE
public int largura(NoArvore<Dados> noAtual)
{
    for (int i = 0; i < 1000; i++)
        quantosNoNivel[i] = 0;
    ContarNosNosNiveis(noAtual, 0);
    return MaiorValorDe(quantosNoNivel);
}

public void ContarNosNosNiveis(NoArvore<Dados> noAtual, int qualNivel)
{
    if (noAtual != null)
    {
        ++quantosNoNivel[qualNivel];
        contarNosNosNiveis(noAtual.Esq, qualNivel+1);
        contarNosNosNiveis(noAtual.Dir, qualNivel+1);
    }
}
```

Exercício 10 – Escrever os antecessores de um determinado nó

```
bool achou = false; // GLOBAL NA CLASSE

public string EscreverAntecessores(NoArvore<Dados> atual, Dado procurado)
{
    string saida = "";
    if (atual != null)
    {
        if (!achou)
            EscreverAntecessores(atual.Esq, procurado);
        if (!achou)
            EscreverAntecessores(atual.Dir, procurado);
        if (atual.Info.CompareTo(procurado) == 0)
            achou = true;
        if (achou)
            saida += "+atual.Info";
    }
    return saida;
}

public string preparaEscritaDosAntecessores(String procurado)
{
    achou = false;
    return EscreveAntecessores(Raiz, procurado);
}
```

6. Desenho da Árvore Binária em um formulário Windows

```
using System.Drawing;

public void DesenharArvore(bool primeiraVez, NoArvore<Funcionario> raiz,
                           int x, int y, double angulo, double incremento,
                           double comprimento, Graphics g)
{
    int xf, yf;
    if (raiz != null)
    {
        Pen caneta = new Pen(Color.Red);
        xf = (int)Math.Round(x + Math.Cos(angulo) * comprimento);
        yf = (int)Math.Round(y + Math.Sin(angulo) * comprimento);
        if (primeiraVez)
            yf = 25;
        g.DrawLine(caneta, x, y, xf, yf);

        DesenharArvore(false, raiz.Esq, xf, yf, Math.PI / 2 + incremento,
                       incremento * 0.60, comprimento * 0.8, g);
        DesenharArvore(false, raiz.Dir, xf, yf, Math.PI / 2 - incremento,
                       incremento * 0.60, comprimento * 0.8, g);

        SolidBrush preenchimento = new SolidBrush(Color.Blue);
        g.FillEllipse(preenchimento, xf - 25, yf - 15, 42, 30);
        g.DrawString(Convert.ToString(raiz.Dados.ToString()), new Font("Comic Sans", 10),
                    new SolidBrush(Color.Yellow), xf - 23, yf - 7);
    }
}
```

Estude atentamente o método acima, da classe Arvore. Ele tem como função desenhar a árvore em um objeto gráfico de um formulário como, por exemplo, um PictureBox. O objeto onde será feito o desenho da árvore é representado pelo parâmetro Graphics g. Graphics é o chamado contexto gráfico, onde se poderá desenhar qualquer coisa. Ou seja, se g se referir a um botão, a árvore será desenhada nesse botão (sem preocupação com proporção da área de desenho e os nós da árvore, veja bem).

O método visita todos os nós da árvore, em percurso que tem ações pré-ordem e pós-ordem; desenha primeiramente as linhas que ligam os nós (pais e descendentes) faz as duas chamadas recursivas e, depois do retorno delas, desenha elipses representando os nós e textos informando o conteúdo do nó.

Temos abaixo um exemplo de uma árvore desenhada usando esse método:

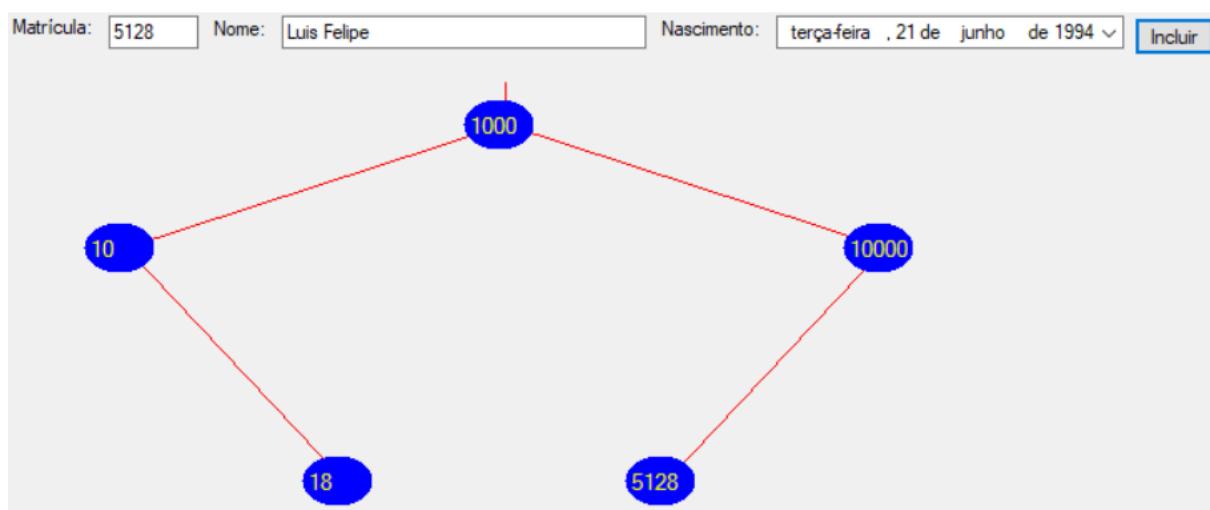


Figura 4 – Uma árvore binária desenhada em um PictureBox de uma aplicação Windows Forms

7. Arquivos de Acesso Direto

Para armazenar dados nas árvores, faremos a leitura das informações a partir de arquivos de registros. Esses arquivos serão um pouco diferentes de arquivos texto, pois seus dados não são dispostos em linhas, e sim são escritos sequencialmente como uma fileira de bytes. Os campos que formam os registros do arquivo terão sempre o mesmo tipo e tamanho, de forma que o tamanho do registro é fixo, não varia de registro para registro. Os dados não serão gravados como sequências de caracteres, mas sim como dados binários de acordo com seu tipo. Isso facilitará algumas operações, como o acesso a qualquer registro diretamente, sem que tenhamos de ler os anteriores.

Algumas linguagens de programação, como Delphi e COBOL, possuem a ideia de arquivos de registros, e comandos que tornam a leitura e escrita dos mesmos bastante simplificada. Basta declarar como o registro é composto (de forma parecida com os atributos de uma classe) e o programa se encarregará de ler os registros, separar os campos de acordo com sua declaração e posição, bem como escrever os dados no arquivo de forma bastante simples.

Já C#, Java e outras linguagens não implementam a ideia de registros ou de arquivos estruturados por registros, e sim somente a ideia de um arquivo sendo um fluxo de bytes.

Por uma questão de eficiência de acesso, será importante que possamos acessar qualquer parte do arquivo diretamente, sem que sejamos obrigados a percorrê-lo sequencialmente, como temos feito com arquivos até o momento. Em Delphi, o acesso a arquivos de registro permite que digamos qual o número de registros que desejamos ler (ou gravar), através de um método de posicionamento, e poderemos acessar o registro desejado, sem termos que passar pelos registros anteriormente armazenados. Isso torna o processo mais rápido.

Mas C# e Java não implementam registros nem arquivos de registros, diferentemente do que ocorre em Delphi.

Para realizar essas operações em C# ou Java, teremos que **modelar classes** que simulem registros e arquivos de registros, com métodos para leitura e para escrita a partir da posição do arquivo em que cada registro se iniciaria.

Os registros lidos serão armazenados em árvores binárias, por meio de métodos de posicionamento dos nós que ainda estudaremos. Nossa árvore será **independente** do tipo de registro que seus nós armazenam. No entanto, esses registros terão de ser modelados de forma a **aderirem às restrições das classes NoArvore e Arvore**.

Nosso objetivo é criar uma maneira de ler dados estruturados armazenados em arquivo. Já sabemos fazer isso para arquivos texto, mas esse tipo de arquivo não permite **acesso direto**, neles somente podemos ler e escrever sequencialmente e isso tornaria ineficiente qualquer uma das populares aplicações de acesso instantâneo e que possuam grandes quantidades de informações, como sistemas de reservas de voos, sistemas bancários, sistemas de ponto de venda, lojas virtuais, dentre outras.

A eficiência e rapidez de acesso são importantes nesses sistemas. Imagine o caso de um sistema de caixa bancário eletrônico: o banco possui milhares ou mesmo milhões de clientes, cujos dados são armazenados em arquivos que, pela própria quantidade de registros, são extensos. No entanto, quando um correntista individual usa um caixa eletrônico, o saldo da sua conta é verificada em poucos segundos, pois o acesso a esse cliente é direto, sem que se tenha de processar sequencialmente todos os registros anteriores ao desse correntista, como teria de ser feito com arquivos de acesso sequencial.

Por esse motivo, arquivos de acesso aleatório são também conhecidos como arquivos de acesso direto.

Boa parte das linguagens de programação não impõe estrutura a arquivos, portanto as aplicações que usam arquivos de acesso aleatório precisam implementar a capacidade de acessar registros estruturadamente. Talvez a forma mais simples de realizar essa capacidade seja obrigar que todos os registros tenham tamanho fixo, ou seja, tamanho igual para todos os registros do arquivo. O uso de registros de tamanho fixo permite ao programa calcular a posição exata de qualquer registro em relação ao início do arquivo. Esse cálculo é feito em função do número de bytes que forma cada registro (tamanho do registro) e do número do registro desejado.

Os arquivos são sequências de bytes. Cada registro tem um tamanho fixo, ou seja, a mesma quantidade de bytes ocorre para cada registro. Dentro da sequência de bytes que forma cada registro, os campos de informação são representados por agrupamentos (subsequências) de bytes que armazenam informações que, tratadas em grupo, dão significado às informações.

Por exemplo, num arquivo de funcionários de uma empresa, cada registro poderia ter os campos abaixo:

Campo	Tipo	Tamanho em bytes
Matrícula	int	4
Nome	string	30
Nascimento	Datetime	8
Código da seção	int	4
Matrícula do chefe	int	4
Quantos dependentes	int	4
Salário	double	8
Em afastamento	bool	1
Tamanho total		63

A figura abaixo ilustra como os registros de tamanho fixo são dispostos nesse arquivo com acesso direto.

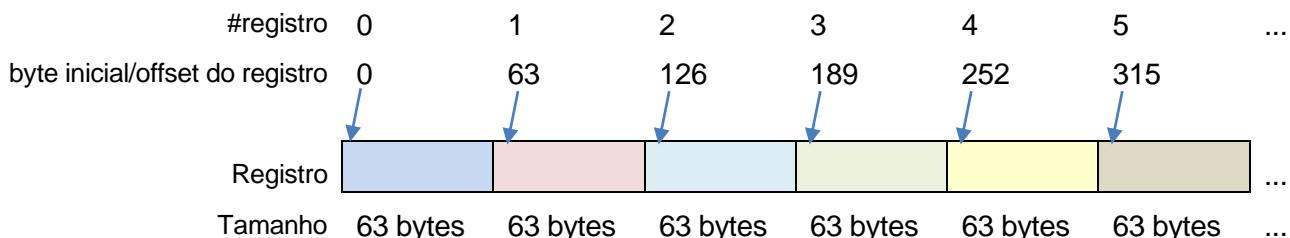


Figura 5 – Disposição dos registros de tamanho fixo em um arquivo de acesso direto/aleatório

Os bytes de um arquivo binário são numerados de 0 a $n-1$, onde n é a quantidade total de bytes do arquivo. Para nossa estruturação por registros, também numeraremos os registros de 0 a $m-1$, onde m é a quantidade de registros armazenados no arquivo. O arquivo binário possui um ponteiro (interno) do byte a ser lido, que pode ser posicionado em qualquer posição válida do arquivo e avança automaticamente conforme leituras e escritas vão sendo realizadas.

Na figura acima, cada registro tem 63 bytes e logo que um registro termina, outro começa. Dentro dos 63 bytes que formam cada registro, durante a **leitura** de um registro a aplicação teria de separar os dados de cada campo. Já durante a **gravação** de um registro, a aplicação teria que agrupar os campos e garantir que sejam escritos na ordem acima (a mesma da leitura), com o tamanho fixo de cada campo, para que leituras seguintes sejam bem sucedidas.

Cada tipo específico de registro específico (alunos, funcionários, livros, rotas, etc.) precisará de classes que leiam, a partir de um arquivo em disco, os campos do registro específico, escrevam-nos nesse arquivo e possam posicionar a leitura no início de cada registro desejado no arquivo.

Para obrigar que cada classe de armazenamento de registros tenha esses métodos, teremos de criar uma Interface que especifique os métodos necessários e fazer com que o tipo genérico Dado adira a essa interface, que chamaremos de **IRegistro**, e ela tem o código abaixo:

```
using System;
using System.IO;
interface IRegistro
{
    void LerRegistro(BinaryReader arquivo, long qualRegistro);
    void GravarRegistro(BinaryWriter arquivo);
    int TamanhoRegistro { get; }
}
```

Qualquer classe de registro que formos armazenar nos nós da árvore binária deverá aderir a essa interface e implementar seus componentes.

O método **LerRegistro** especifica qual leitor de arquivo será usado e o número do registro que se deseja ler. Esse método, ao ser implementado por uma classe de registro, deverá ler cada campo separadamente, byte a byte, e organizar os atributos da classe para que recebam as informações lidas.

A classe **BinaryReader** pertence ao namespace System.IO e é associada ao arquivo binário que será aberto para leitura. Esse arquivo será do tipo **FileStream**, e o objeto da classe FileStream ficará a cargo da classe que usa essa interface para ser associado a um arquivo físico e instanciado.

Em seguida, BinaryReader permite **posicionar a leitura** de um arquivo **em qualquer um de seus bytes**, ou seja, se você souber onde se inicia o registro que deseja ler, fazemos um posicionamento na posição inicial (no byte inicial) desse registro e o objeto BinaryReader poderá ler seus campos.

O método **GravarRegistro** especifica qual escritor de arquivo será usado. Ao ser implementado por uma classe de registro, esse método deverá escrever no arquivo campo a campo, no tamanho adequado, a partir dos dados dos atributos da classe.

A gravação não especifica o número do registro desejado porque os dados serão gravados na **sequência** em que ocorrer o **percurso na árvore binária**.

A classe **BinaryWriter** funciona de maneira semelhante a BinaryReader, mas serve para escrever dados em um fluxo de bytes (stream) de saída, ou seja, em um arquivo binário de saída.

A propriedade TamanhoRegistro é usada no cálculo da posição física do arquivo em que se iniciará a leitura de um registro.

Usando como base a descrição do arquivo de funcionários acima, podemos codificar uma classe Funcionario, que possa ser encapsulada por um NoArvore e, também, atenda às exigências da interface IRegistro. Assim, os atributos da classe seriam:

```
using System;
using System.IO;
using System.Windows.Forms;

class Funcionario : IComparable<Funcionario>, IRegistro
{
    public const int tamanhoNome = 30;

    private int matricula;
    private string nome;
    DateTime nascimento;
    int codigoSecao;
    int matriculaChefe;
    int quantosDependentes;
    double salario;
    bool afastado;
```

Precisamos saber o tamanho desse registro e, para isso, declaramos a constante inteira abaixo:

```
const int tamanhoRegistro = sizeof(int)+          // matricula
                           tamanhoNome+           // nome
                           sizeof(Int64)+         // nascimento
                           sizeof(int)+           // codigoSecao
                           sizeof(int)+           // matriculaChefe;
                           sizeof(int)+           // quantosDependentes
                           sizeof(double)+        // salario
                           sizeof(bool);          // afastado
```

Os campos string de nossos arquivos de registro precisam ter um tratamento especial, pois **strings** em C#, Java e outras linguagens **não possuem tamanho fixo**. Por isso declaramos a constante

tamanhoNome, valendo 30 e, para garantir que a string tenha o tamanho de 30 caracteres (nem mais nem menos), usaremos a propriedade abaixo para delimitar o tamanho e preencher a string com espaços à direita, se for necessário:

```
public string Nome
{
    get => nome;
    set => nome = value.PadRight(tamanhoNome, ' ').Substring(0, tamanhoNome);
}
```

Se a string armazenada na propriedade Nome tiver mais de 30 caracteres, não haverá preenchimento com espaços e, logo em seguida, será truncada pelo Substring. Ela nunca ultrapassará 30 caracteres.

Abaixo temos as demais propriedades, para permitir o acesso aos atributos (campos de dados) da classe que modela o registro de funcionários:

```
public int Matricula { get => matricula; set => matricula = value; }
public DateTime Nascimento { get => nascimento; set => nascimento = value; }
public int CodigoSecao { get => codigoSecao; set => codigoSecao = value; }
public int MatriculaChefe { get => matriculaChefe; set => matriculaChefe = value; }
public int QuantosDependentes { get => quantosDependentes; set => quantosDependentes = value; }
public double Salario { get => salario; set => salario = value; }
public bool Afastado { get => afastado; set => afastado = value; }
```

Para que possamos instanciar um objeto de um tipo **genérico** em uma outra classe, precisamos que o tipo **específico** implemente um **construtor default, sem parâmetros**. Vimos isso em Técnicas de Programação I, quando estudamos a classe `VetorDados<Dado>`, onde havia a restrição `where Dado : new()`, que torna obrigatória a implementação de um construtor sem parâmetros.

O construtor sem parâmetros precisa existir porque é a classe Arvore que instanciará um objeto da classe genérica Dado, mas Arvore não conhece os detalhes internos da classe específica que **substituirá Dado na aplicação**. Assim, a classe Arvore passará a ter a declaração abaixo:

```
class Arvore<Dado> where Dado : IComparable<Dado>, IRegistro, new()
```

Isso ainda não tinha sido feito porque não instanciamos os dados que serão armazenados na árvore ainda e, fazendo assim, deixamos que essa instanciação seja o mais genérica possível e **feita na classe Arvore** e não na aplicação. Assim, a classe Arvore poderá **delegar para a classe específica** que leia o arquivo e acesse os campos de acordo com suas especificidades.

Vamos então criar o construtor default da classe Funcionario:

```
public Funcionario()
{
    Matricula = 0;
    Nome = "";
    CodigoSecao = 0;
    MatriculaChefe = 0;
    QuantosDependentes = 0;
    Salario = 0.0f;
    Afastado = false;
}
```

Aproveitemos e vamos também criar um construtor parametrizado, com os dados de todos os campos do registro, e um construtor com apenas a matrícula, que é o campo chave primária desse registro:

```
public Funcionario(int matricula, string nome, DateTime nascimento, int codigoSecao,
                   int matriculaChefe, int quantosDependentes, float salario, bool afastado)
{
```

```
this.Matricula = matricula;
this.Nome = nome;
this.Nascimento = nascimento;
this.CodigoSecao = codigoSecao;
this.MatriculaChefe = matriculaChefe;
this.QuantosDependentes = quantosDependentes;
this.Salario = salario;
this.Afastado = afastado;
}
public Funcionario(int matricula)
{
    this.matricula = matricula;
}
```

Como essa classe também tem que implementar CompareTo, dada a restrição IComparable, codificamos abaixo esse método, usando a matrícula como campo chave de comparação:

```
public int CompareTo(Funcionario outroFunc)
{
    return matricula - outroFunc.matricula;
}

public override string ToString()
{
    return Matricula+ "";
}
```

Observe que essas operações que estamos criando poderiam ser usadas em qualquer estrutura de dados que tivesse a origem de seus dados a partir de um arquivo binário de acesso direto, organizado por registros para uma aplicação específica de acesso e manutenção desse tipo de registro. Vamos usar essas ideias logo em seguida, para buscar dados para armazenar em nossa árvore e, também, para ordená-la, tornando-a uma estrutura bastante eficiente para armazenamento e recuperação de informações, como se fosse um vetor ordenado com pesquisa binária para buscar seus dados.

Em seguida, vamos passar a implementar, na classe Funcionario, a propriedade e os métodos exigidos pela interface IRegistro:

```
public int TamanhoRegistro { get => tamanhoRegistro; }
public void LerRegistro(BinaryReader arquivo, long qualRegistro)
{
    if (arquivo != null)
        try
    {
```

Calculamos o offset (deslocamento) de bytes desde o início do arquivo que nos posiciona exatamente no início do registro desejado:

```
long qtosBytes = qualRegistro * TamanhoRegistro;
```

Posicionamos a leitura na posição do byte calculado acima, usando o início do arquivo como ponto de partida:

```
arquivo.BaseStream.Seek(qtosBytes, SeekOrigin.Begin);
```

Arquivo.BaseStream acessa o FileStream associado ao arquivo binário que está sendo tratado. O método **Seek** posiciona o ponteiro de acesso ao arquivo exatamente no byte informado no seu primeiro parâmetro, contado a partir da origem de busca indicado (Begin para acesso a partir do byte

0 do arquivo – início do arquivo, e End para acesso a partir do último byte do arquivo – acesso do final para o início).

Fazemos a leitura dos campos conforme sua descrição (inteiros, reais, lógicos, strings) e na sequência em que os campos deverão ser gravados. **ReadInt32()** lê a quantidade de bytes que formam um inteiro de 4 bytes, que é o tamanho ocupado pelo campo inteiro matricula. Armazenamos na propriedade Matricula para que ela possa processar o dado armazenado caso isso tenha sido nela codificado.

```
this.Matricula = arquivo.ReadInt32();
```

Para campos string, devemos primeiramente armazená-los em um vetor de caracteres, de tamanho igual ou superior ao tamanho de cada campo. Por esse motivo, dentre outros, declaramos a constante tamanhoNome para informar a aplicação e a classe sobre quantos caracteres a string Nome deverá ter obrigatoriamente:

```
char[] umNome = new char[tamanhoNome];
```

Fazemos a leitura dos caracteres que formam a string nome, usando o método ReadChars, com um parâmetro que indica quantos caracteres deverão ser lidos em sequência no arquivo. Os bytes lidos serão armazenados, em sequência, nas posições do vetor umNome.

```
umNome = arquivo.ReadChars(tamanhoNome);
```

Em seguida, copiamos cada um desses caracteres, um a um, na variável string nomeLido e, ao final dessa cópia, armazenamos nomeLido na propriedade Nome para que ela possa processar o dado armazenado caso isso tenha sido nela codificado (por exemplo, truncar, preencher com espaços, com zeros, dependendo do que seja necessário). Para campos string esse é o procedimento de leitura.

```
9         string nomeLido = "";
10        for (int i=0; i<tamanhoNome; i++)
11            nomeLido += umNome[i];
12        Nome = nomeLido;
```

Em seguida, lemos **um inteiro de 8 bytes**, pois datas são armazenadas como uma quantidade de Ticks que representa a data de nascimento. Um único **tick** representa um centésimo de nanosegundo, ou um décimo-milionésimo de um segundo. O valor dessa propriedade representa a quantidade de intervalos de 100 nanossegundos de duração que ocorreram desde as 12:00:00 meia noite, de 1 de Janeiro de 0001 no calendário Gregoriano (valor de DateTime.MinValue).

```
Int64 dadoData = arquivo.ReadInt64();
try
{
    Nascimento = new DateTime(dadoData);
}
catch (Exception e1)
{
    MessageBox.Show(e1.Message+" "+dadoData);
}
```

Em seguida, leremos os demais campos, cada um de acordo com o tipo de dado e a quantidade de bytes que formam cada campo, usando os métodos ReadInt32(), ReadDouble() e ReadBoolean().

```
CodigoSecao = arquivo.ReadInt32();
MatriculaChefe = arquivo.ReadInt32();
QuantosDependentes = arquivo.ReadInt32();
```

```
        Salario = arquivo.ReadDouble();
        Afastado = arquivo.ReadBoolean();
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
    }
}
```

Observe que, conforme fazemos a leitura dos dados, o ponteiro do byte a ser lido **vai avançando na estrutura do registro**.

Já no método de gravação, como ela será sequencial, podemos escrever os dados diretamente em sequência. Usamos um BinaryWriter associado a um FileStream que, por sua vez, indica o arquivo físico onde se deseja gravar os registros.

Usamos o método Write para gravar os dados de cada campo. O tamanho usado para a gravação de cada campo depende somente do tipo do campo a ser gravado. Por esse motivo não temos WriteInt32, WriteDouble, WriteChars, WriteBoolean, somente temos Write que se adapta ao tamanho físico de cada campo que se deseja gravar no arquivo. Os bytes são gravados e o ponteiro de acesso ao arquivo binário automaticamente avança para a posição seguinte à que foi gravada por último.

Observe, no código abaixo, que novamente tratamos o campo string **nome** como um vetor de **char**, para que o método **Write** saiba como gravá-lo. Quando Write tem um vetor de char como parâmetro, grava exatamente a mesma quantidade de caracteres que o número físico de posições desse vetor:

```
public void GravarRegistro(BinaryWriter arquivo)
{
    if (arquivo != null)
    {
        arquivo.Write(Matricula);
        char[] umNome = new char[tamanhoNome];
        for (int i = 0; i < tamanhoNome; i++)
            umNome[i] = nome[i];
        arquivo.Write(umNome);
        arquivo.Write(Nascimento.Ticks);
        arquivo.Write(CodigoSecao);
        arquivo.Write(MatriculaChefe);
        arquivo.Write(QuantosDependentes);
        arquivo.Write(Salario);
        arquivo.Write(Afastado);
    }
}
```

8. Árvores de Busca

Conceitos Gerais

É possível armazenar registros de arquivos binários de dados, como os que vimos na sessão anterior, em árvores binárias, para manter organizadas as informações, ou seja, em ordem. Essa ordenação permite que se efetuem pesquisas aos dados de maneira muito mais rápida do que o tempo gasto nos percursos estudados até o momento.

O processo consiste em se dispor os dados de uma maneira que, baseando-se no conteúdo de um campo chave para identificação de cada nó, exista uma ordem de colocação pré-estabelecida.

Dado o nó raiz, elementos cuja chave sejam menores que a chave do nó raiz, serão colocados na sub-árvore esquerda, e aqueles cujas chaves forem maiores que a da raiz, serão colocados na sub-árvore direita. Claro está que cada sub-árvore é uma árvore por si só, e portanto a colocação do elemento nas sub-árvores respeita a mesma regra: compara-se a chave do elemento a colocar com a chave da raiz da sub-árvore, colocando esse novo nó à esquerda da "sub-raiz", caso sua chave seja menor, ou à direita, caso possua chave maior que a chave da "sub-raiz". Isso se repete até que se encontre uma folha da árvore original, quando o novo elemento será ligado a essa folha.

Esse tipo de árvore é chamado de árvore de busca, por permitir a realização de pesquisa binária na árvore. Por exemplo:

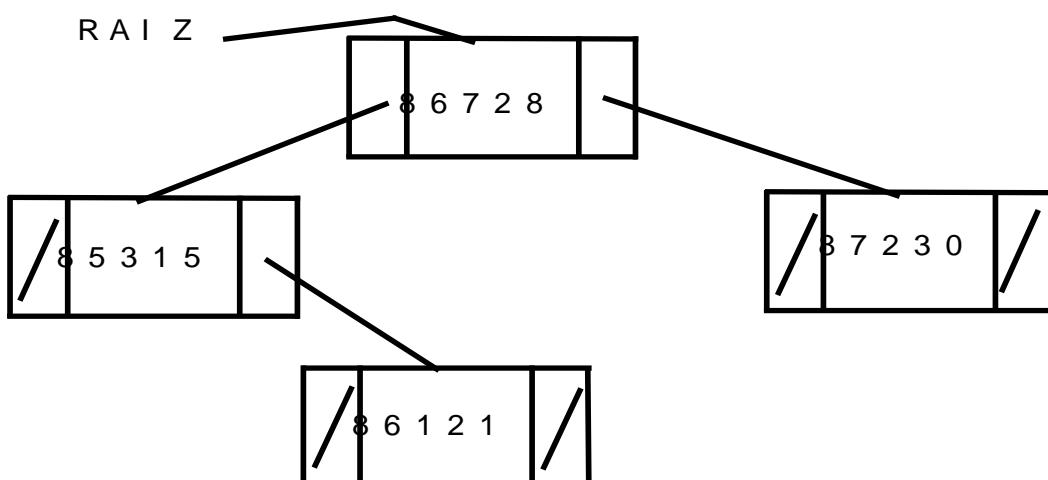


Figura 6 – Árvore binária de busca

1. Se desejamos procurar os dados do aluno cujo RA (chave de acesso) é 85315, devemos procurar à esquerda da Raiz, pois $85315 < 86728$. Em outras palavras, chaves menores que a da Raiz devem estar no ramo esquerdo da árvore. Pode até acontecer de a chave procurada não existir, mas temos certeza de que a mesma não se encontra no ramo direito. Isso já permite ignorar metade da árvore em média, diminuindo o campo de procura.
2. Da mesma forma, o aluno com RA 86121 estará no ramo esquerdo da árvore, pois sua chave é menor que a chave do nó Raiz. Mas, como o ramo esquerdo também é uma árvore, o nó do aluno procurado deverá estar à direita da raiz dessa sub-árvore, que é o nó com RA 85315. Como $86121 > 85315$, o elemento procurado não estará à esquerda da atual Raiz, e sim poderá estar à sua direita.
3. Se o aluno procurado tivesse RA igual a 88431, a procura se daria à direita da raiz, e depois, novamente à direita do nó com chave = 87230. Assim sendo, chegaríamos ao final da árvore, sem que se encontrasse o elemento procurado.

Essa forma de dispor os elementos da árvore é feita para facilitar a pesquisa de um elemento qualquer da árvore, como numa pesquisa binária em vetor, no qual os dados estão dispostos em ordem.

Note que, caso a árvore seja percorrida em **In-Ordem**, teremos como resultado uma lista ordenada crescentemente pelo campo chave (RA). Assim, chegamos à conclusão de que esta é uma técnica de ordenação, usando uma estrutura de dados para alocação dinâmica. Neste método de ordenação, não existe necessidade de se trocar elementos ou deslocá-los fisicamente.

A pesquisa de um elemento é realizada da seguinte maneira:

- Um apontador auxiliar recebe o endereço da Raiz da árvore.
- Se o apontador auxiliar é igual a **null**, a chave não existe.
- A chave procurada é comparada com a chave do nó indicado pelo apontador auxiliar. Se forem iguais, encontrou-se o nó com a chave desejada.
- Se a chave da Raiz for maior que a chave procurada, desloca-se o apontador auxiliar para a raiz da sub-árvore esquerda, e retorna-se ao ponto b.
- Se a chave da Raiz for menor que a chave procurada, desloca-se o apontador auxiliar para a raiz da sub-árvore direita, e retorna-se ao ponto b.

Verifiquemos algumas características desse tipo de árvore, observando a árvore da figura abaixo. Note onde estão o menor e o maior elementos da árvore. O menor de todos é 84200. Para acessá-lo, deve-se colocar um apontador no nó mais à esquerda da árvore. Isso vale para qualquer sub-árvore, de modo que o nó mais à esquerda de qualquer sub-árvore será o com a menor chave, no caso de uma árvore de busca. Já o maior elemento é aquele que fica mais à direita da árvore. Isso também vale para qualquer sub-árvore, já que a definição de uma árvore de busca é recursiva, valendo para qualquer sub-árvore que a componha. Para atingir a maior chave, deve-se percorrer a árvore sempre à direita, até chegarmos a um nó que não tenha descendente à direita. Esse será o nó com a maior chave. O mesmo se dá simetricamente com o menor elemento, no ramo esquerdo.

Como exemplo, vamos estudar os passos para encontrar as chaves 85127 e 87220 na árvore da figura 7, abaixo. O endereço de cada nó é dado pelo número inteiro colocado sobre cada nó.

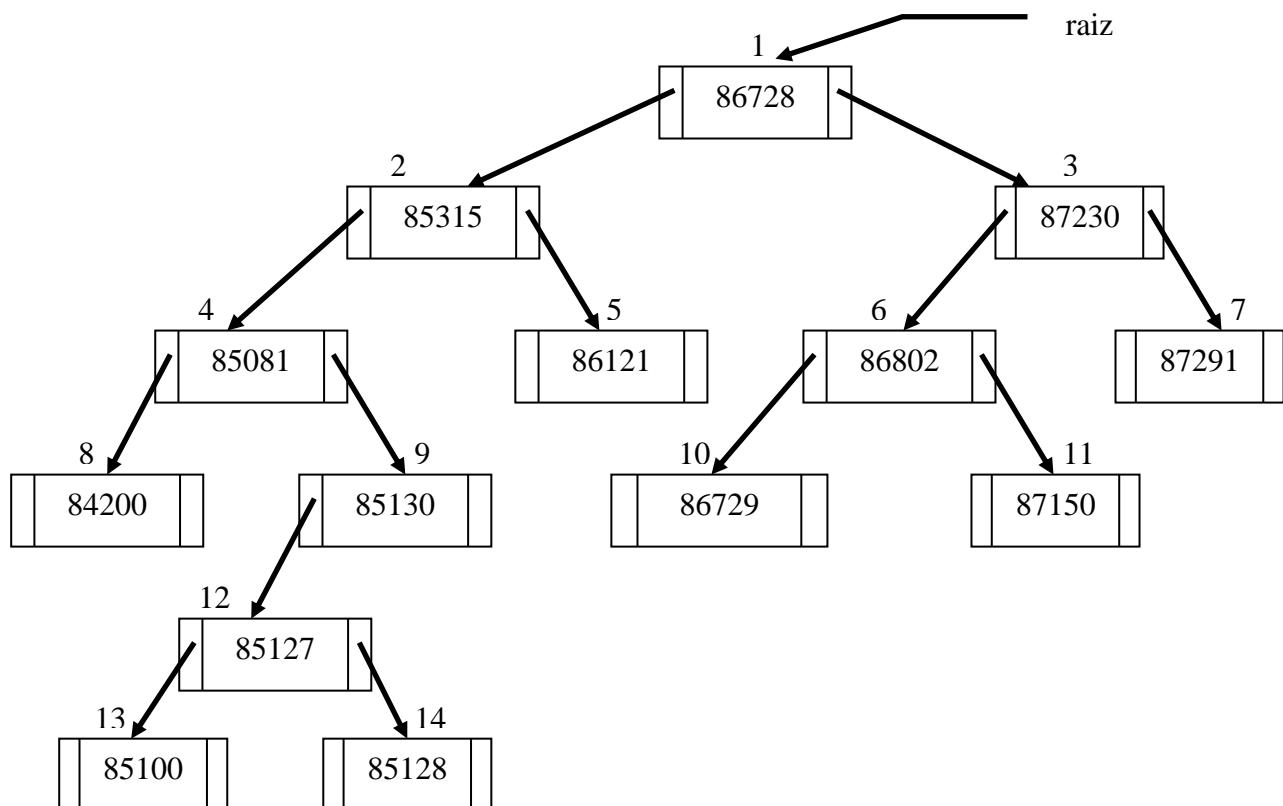


Figura 7– Árvore binária de busca para estudo de pesquisa binária

- Comparamos 85127 com a chave da Raiz 1. Como 85127 é **menor** que 86728, deve-se deslocar o apontador de pesquisa para o descendente esquerdo do nó 1 (ou seja, para o nó 2).

2. Comparamos agora 85127 com a chave do nó **2** (85315). Como 85127 é **menor** que 85315, o nó procurado só pode estar à esquerda do nó atual, logo o apontador de pesquisa deve ser deslocado para o descendente esquerdo do nó atual, passando assim a apontar para o nó 4.
3. Como 85127 é **maior** que 85081, a chave procurada só pode estar à direita do nó atual. Assim, o ponteiro é deslocado para o descendente direito do nó atual, indo da posição 4 para 9.
4. Como 85127 é **menor** que 85130, a chave procurada só poderia estar à esquerda do nó atual, e portanto deslocamos o apontador do nó **9** para o nó **12**.
5. Como 85127 (chave do nó 12) é **igual** à chave procurada, o nó onde se encontra a chave desejada foi encontrado, e pode-se então realizar-se a operação desejada, seja ela uma inclusão, uma exclusão, exibição, etc.

Note que o número de acessos foi menor do que ocorreria se usássemos uma lista ligada. Note também que esse tipo de pesquisa é uma pesquisa binária, pois a cada decisão descartamos metade dos nós ainda não descartados da árvore.

Já no caso da chave 87220, que não se encontra na árvore, teríamos o seguinte percurso:

1. Comparamos 87220 com a chave da Raiz **1**; como 87220 é **maior** que 86728, o apontador é deslocado à direita da Raiz.
2. Comparamos 87220 com 87230, a chave do nó 3. Como 87220 é menor que 87230, desloca-se o apontador à esquerda, para o nó **6**.
3. Verificamos que 87220 é maior que 86802 (chave do nó **6**), e assim o apontador deve ser deslocado à direita, indo para o nó **11**.
4. Neste momento, comparamos 87220 com 87150, e o apontador é deslocado à direita, passando a valer **null**.
5. Como o apontador chegou a **null**, a chave procurada não existe na árvore.

Note que este fato faz com que uma árvore vazia (**raiz == null**) produza como resultado de pesquisa a não existência da chave procurada.

Um algoritmo de pesquisa recursiva é facilmente montado. Note que as saídas são duas: (a) quando o apontador indica **null**, o elemento procurado não existe; (b) quando o apontador indica um nó que contém a chave procurada. Caso nenhuma dessas condições ocorra, deve-se verificar para qual ramo o apontador será deslocado. Esse deslocamento é feito por meio da passagem do apontador para a esquerda ou para a direita, numa chamada recursiva, como parâmetro para a próxima instância da pesquisa:

```
public bool ExisteRec(NoArvore<Dado> local, Dado procurado)
{
    if (local == null)
        return false;
    else
        if (local.Info.CompareTo(procurado) == 0)
        {
            atual = local;
            return true;
        }
        else
        {
            antecessor = local;
            if (procurado.CompareTo(local.Info) < 0)
                return ExisteRec(local.Esq, procurado); 1 // Desloca apontador na
                                                       // próxima instância do
            else
                return ExisteRec(local.Dir, procurado); 2 // método
        }
}
```

O ponteiro **atual**, da classe Arvore, receberá o endereço do nó onde foi encontrada a chave. No entanto, como discutimos anteriormente, sempre que for possível evitar a recursão, é melhor fazê-lo. Esse

algorítmo é realmente um daqueles em que se pode evitar a recursão, por meio do deslocamento do apontador através de atribuição, como se mostra abaixo:

```
public bool Existe(Dado procurado)
{
    antecessor = null;
    atual = raiz;
    while (atual != null)
    {
        if (atual.Info.CompareTo(procurado) == 0)
            return true;
        else
        {
            antecessor = atual;
            if (procurado.CompareTo(atual.Info) < 0)
                atual = atual.Esq;      // Desloca apontador para o ramo à esquerda
            else
                atual = atual.Dir;    // Desloca apontador para o ramo à direita
        }
    }
    return false;           // Se local == null, a chave não existe
}
```

A partir deste momento, estamos aptos a discutir a montagem (criação) de árvore de busca, quais os problemas que podem ocorrer nesse processo e como solucioná-los.

9. Montagem inicial de uma árvore binária de busca

A montagem inicial da árvore de busca terá dados lidos, geralmente, a partir de um arquivo em disco. Usaremos as ideias que desenvolvemos sobre arquivos binários de acesso direto aqui. A criação de uma árvore de busca deve, obviamente, prever a colocação dos elementos na disposição descrita nos [Conceitos Gerais](#). Assim sendo, teremos duas partes principais nesse processo:

1. Leitura dos dados a incluir
2. Colocação de cada registro lido na posição correta na árvore binária de busca.

O item 1 nada mais é do que uma repetição em que se efetua a leitura dos dados do arquivo em disco, desde o primeiro até o último registro. Este processo será responsável pela chamada do processo de colocação do nó na árvore (item 2).

```
var dado = new Dado();           // cria um objeto para armazenar os dados
int quantosRegistros = arquivo.BaseStream.Length / dado.TamanhoRegistro;
for (int registroDesejado = 0; registroDesejado < quantosRegistros; registroDesejado++)
{
    dado.LerRegistro(arquivo, registroDesejado); // percorre sequencialmente o arquivo
    ... // chamará a inclusão na árvore aqui (item 2)
}
```

Dependendo de como for implementado, o item 2 pode ser usado sempre que for necessário incluir-se um nó na árvore, e não apenas na montagem inicial da mesma. Em linhas gerais, o item 2 tem o seguinte algoritmo:

- a. Se o apontador atual da árvore for **null**, cria-se o nó para o novo registro e guarda-se seu endereço no apontador atual.
- b. Se o apontador atual for diferente de **null**, então :

1. Se a chave do registro lido for igual à chave do nó atualmente apontado, então o registro já existe, e alguma ação corretiva poderá ser necessária.
2. Se as chaves não forem iguais, verifica-se em que ramo da árvore dever-se-á incluir o novo registro. Se a chave do novo registro for menor do que a chave do nó atualmente apontado, então o novo registro deverá ser posicionado no ramo esquerdo, derivado do nó atual. Se, por outro lado, a chave do novo registro for maior do que a chave do nó atual, então deve-se posicionar o novo registro no ramo direito derivado do nó atual.

Note que se for necessário dirigir-se a um dos ramos derivados do nó atual, existirá uma recorrência dos passos acima. Em outras palavras, se decidimos que o novo nó deveria estar à direita do nó atual (chave nova maior que chave atual), isso indica que o novo nó deverá ser um descendente ligado ao ramo direito que deriva do nó atual. Esse ramo direito tem como apontador o campo `atual.Dir`.

Se usarmos recursão para implementar o algoritmo, então deveremos passar esse campo como parâmetro, para que seja o apontador do nó atual da próxima instância recursiva. Abaixo está um trecho do algoritmo:

```
public void Incluir(Dado dadoLido)
{
    Incluir(ref raiz, dadoLido);
}

private void Incluir(ref NoArvore<Dado> atual, Dado dadoLido)
{
    if (atual == null)
    {
        atual = new NoArvore<Dado>(dadoLido);
    }
    else
        if (dadoLido.CompareTo(atual.Info) == 0)
            throw new Exception("Já existe esse registro!");
```

Como se pode notar, passa-se como parâmetro um apontador para o nó atual. Na primeira chamada, o argumento passado será o apontador da **raiz** da árvore; esse apontador deverá começar com **null** no início do programa.

Quando isso acontecer, um novo nó será criado, e seu campo `Info` será preenchido com os dados lidos na variável `dadoLido`. Note também que os apontadores para os ramos esquerdo e direito desse novo nó são preenchidos, no **construtor** chamado de `NoArvore`, com **null**, indicando que não há descendentes. Isso é bastante importante, pois é essa atribuição de nulo a esses apontadores de ramos esquerdo e direito que garantirá a correta ligação entre os descendentes da raiz.

Observe também que, quando as chaves são iguais, uma exceção é disparada. O procedimento a tomar nesse caso depende exclusivamente dos requisitos da aplicação, que deveria capturar exceções ao usar esse método. Uma mensagem ao usuário poderia ser emitida, por exemplo, e a inclusão finalizada.

Agora, pense no caso de inclusão de um novo nó cuja chave é maior que a da **raiz**. Lemos o registro `dadoLido`, e chamamos o procedimento acima. Como atual agora não é mais **null** (pois na primeira chamada passamos `raiz` como argumento), o fluxo de execução passa para a comparação da chave do registro lido com a do nó apontador por `atual` (que por ser a primeira chamada, é a raiz da árvore). Como definimos que a nova chave é maior que a da raiz, então teremos de nos dirigir à direita da árvore. Para isso, passamos o apontador do ramo direito do nó atual como argumento de chamada, como vemos abaixo:

```
private void Incluir(ref NoArvore<Dado> atual, Dado dadoLido)
{
    if (atual == null)
        atual = new NoArvore<Dado>(dadoLido); // Construtor anula esq e dr
```

```
        else
            if (dadoLido.CompareTo(atual.Info) == 0)
                throw new Exception("Já existe esse registro!");
        else
            if (dadoLido.CompareTo(atual.Info) > 0)
            {
                NoArvore<Dado> apDireito = atual.Dir;
                Incluir(ref apDireito, dadoLido);
                atual.Dir = apDireito;
            }
```

C# não permite que propriedades sejam passadas por referência e, por esse motivo, passamos como parâmetro a variável auxiliar `apDireito`, que havia antes recebido o valor de `atual.Dir`. Após o fluxo de execução retornar dessa chamada recursiva, atualizamos o valor de `atual.Dir` com o possível novo valor de `apDireito`.

Uma variável passada pelo `out` ou `ref` só pode ser uma variável de verdade. Uma propriedade não é uma variável, parece ser uma mas é um método que é chamado para possivelmente acessar uma variável de forma indireta. Os modificadores `out` e `ref` acabam tendo uma [indireção](#) e por isso são passadas como um ponteiro e a semântica esperada é que isto seja um ponteiro para um dado de forma direta. Se tiver uma nova indireção o compilador não sabe o que fazer.

Se quiser entender mais sobre como funciona esse modificador tem resposta sobre em [O que são os parâmetros out e ref](#). E para saber mais sobre propriedade tem mais em [Como funcionam as propriedades no C#? \(mais e mais\)](#).

Fonte: [https://pt.stackoverflow.com/questions/417048/erro-uma-propriedade-ou-um-indexador-que-não-possa-ser-passado-como-um-parâmet](https://pt.stackoverflow.com/questions/417048/erro-uma-propriedade-ou-um-indexador-que-n%C3%A3o-possa-ser-passado-como-um-par%C3%A3met)

Analizando o que poderá ocorrer agora, notamos que na próxima instância do procedimento `atual` receberá como argumento o endereço do campo `atual.dirita`. Como estamos supondo que a árvore só tem o primeiro nó, então `atual`, neste nova instância, valerá `null`. Mas quando `atual` vale `null`, o procedimento cria um novo nó e coloca seu endereço em `atual`. No entanto, `atual` refere-se ao campo `direita` do nó anterior, pois o parâmetro `atual` contém seu endereço e, assim, é esse campo que receberá o endereço alocado. Logo, o campo `Direita` do nó Raiz (Atual na instância anterior) apontará para o novo nó, cujas informações são as do novo registro lido. O mesmo aconteceria se o ramo fosse o da esquerda, como podemos notar no algoritmo completo abaixo:

```
private void Incluir(ref NoArvore<Dado> atual, Dado dadoLido)
{
    if (atual == null)
        atual = new NoArvore<Dado>(dadoLido);
    else
        if (dadoLido.CompareTo(atual.Info) == 0)
            throw new Exception("Já existe esse registro!");
        else
            if (dadoLido.CompareTo(atual.Info) > 0)
            {
                NoArvore<Dado> apDireito = atual.Dir;
                Incluir(ref apDireito, dadoLido);
                atual.Dir = apDireito;
            }
            else
            {
                NoArvore<Dado> apEsquerdo = atual.Esq;
                Incluir(ref apEsquerdo, dadoLido);
                atual.Esq = apEsquerdo;
            }
}
```

Sempre que um novo nó é criado, seus apontadores `Esp` e `Dir` recebem `null`, no construtor da classe `NoArvore<Dado>()`. É essa atribuição que assegura que quando um nó derivado desse novo nó for

necessário, atual valerá **null**. Como a passagem do parâmetro é por referência, qualquer modificação dentro de uma instância é propagada para as instâncias anteriores, e assim, quando se efetua **new**, quem recebe o endereço alocado é o argumento passado na chamada. Dessa maneira, a ligação entre os nós é realizada, e a árvore é montada, como podemos notar pelo diagrama de execução abaixo. Suponha que estamos incluindo a chave 87170:

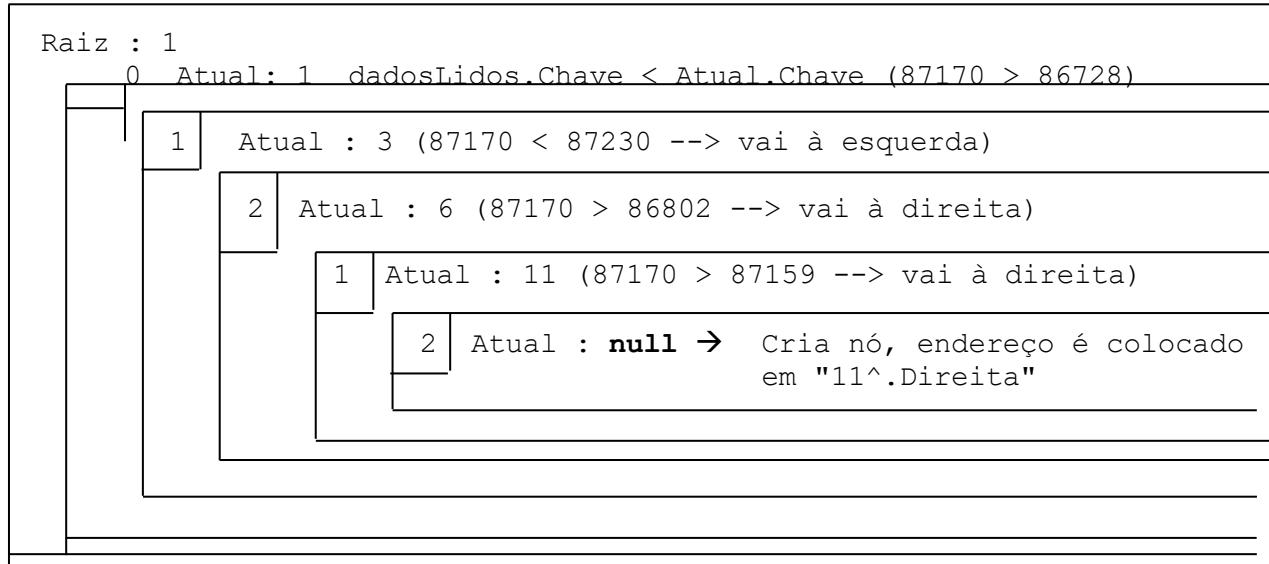


Figura 8 – Diagrama de execução do método recursivo *Incluir()*

Esse método *Incluir()* consiste em uma versão recursiva da inclusão de nós em uma árvore de busca. Compare-o com o método de [pesquisa binária recursiva na árvore](#). Veja que são bem parecidos em suas abordagens, passando parâmetros através de chamadas recursivas para, assim, navegar na árvore binária até achar o elemento procurado (ou o local de sua inclusão, no caso do método *Incluir()*).

Da mesma maneira, temos um método de [pesquisa binária iterativa \(não-recursiva\)](#). Sabemos que, quando for possível evitar a recursão, é melhor fazê-lo. Assim, podemos usar esse método de pesquisa binária não-recursiva (menor gasto de tempo e de memória) como base para criarmos um novo método de Inclusão, que vem abaixo:

```

public void Inserir(Dado novosDados)
{
    bool achou = false, fim = false;
    NoArvore<Dado> novoNo = new NoArvore<Dado>(novosDados);
    if (raiz == null)           // árvore vazia
        raiz = novoNo;
    else                      // árvore não-vazia
    {
        antecessor = null;
        atual = raiz;
        while (!achou && !fim)
        {
            antecessor = atual;
            if (novosDados.CompareTo(atual.Info) < 0)
            {
                atual = atual.Esq;
                if (atual == null)
                {
                    antecessor.Esq = novoNo;
                    fim = true;
                }
            }
        }
    }
}
  
```

```
        if (novosDados.CompareTo(atual.Info) == 0)
            achou = true; // pode-se disparar uma exceção neste caso
        else
        {
            atual = atual.Dir;
            if (atual == null)
            {
                antecessor.Dir = novoNo;
                fim = true;
            }
        }
    }
}
```

Esse método busca o local de inclusão em um nível inferior da árvore, sempre sabendo quem é o antecessor do nó que será inserido (o pai do novo nó). Temos um caso especial para o primeiro nó da árvore, quando a raiz é nula e, em seguida, caso a raiz não seja nula, percorre-se a árvore de busca como se estivéssemos procurando o elemento a ser inserido, usando o ponteiro atual para isso. Antecessor sempre aponta o nó ancestral de atual. Quando o percurso leva a um ponteiro nulo (atual fica valendo nulo), encontramos o local de inclusão e, nesse caso, ligamos o antecessor ao novo nó, que passa a ser um descendente esquerdo ou direito, dependendo do ramo pelo qual o percurso seguiu.

Mas, também, podemos usar o método de busca binária (Existe()) para percorrer a árvore e **descobrir o local de inclusão**. Já temos esse método feito e, portanto, usá-lo como auxiliar da inclusão tornaria o código mais reutilizável e menor. Tanto o método de busca binária não-recursivo quanto a versão recursiva desse método percorrem a árvore até encontrar o elemento procurado (caso que não nos serve numa inclusão) ou até encontrar um descendente nulo do **antecessor** (**atual** ficou com nulo e nesse local devemos incluir o novo nó). Quando não encontramos o registro procurado, o atributo **antecessor** da classe ArvoreDeBusca aponta o pai desse registro, caso ele fosse incluído. Portanto, toda a parte de busca do local onde ficaria o novo nó a ser incluído pode ser realizada pelos métodos Existe() ou ExisteRec().

Essa nova e última versão da inclusão vem abaixo:

```
public void IncluirNovoRegistro(Dado novoRegistro)
{
    if (Existe(novoRegistro))
        throw new Exception("Registro com chave repetida!");
    else
    {
        // o novoRegistro tem uma chave inexistente, então criamos um
        // novo nó para armazená-lo e depois ligamos esse nó na árvore
        var novoNo = new NoArvore<Dado>(novoRegistro);

        // se a árvore está vazia, a raiz passará a apontar esse novo nó
        if (raiz == null)
            raiz = novoNo;
        else
            // nesse caso, antecessor aponta o pai do novo registro e
            // verificamos em qual ramo o novo nó será ligado
            if (novoRegistro.CompareTo(antecessor.Info) < 0) // novo é menor que antecessor
                antecessor.Esq = novoNo; // vamos para a esquerda
            else
                antecessor.Dir = novoNo; // ou vamos para a direita
    }
}
```

Em uma aplicação Windows Forms, podemos ter um evento Click de um botão que inclua registros na árvore. No trecho de código abaixo, supomos que temos uma aplicação de manutenção de dados de funcionários, como o registro que discutimos na seção sobre Arquivos de Acesso Direto. Criamos uma instância da [classe Funcionario](#), usando um de seus [construtores](#) com os dados digitados em campos do formulário Windows e a enviamos como parâmetro para o método IncluirNovoRegistro, que acabamos de codificar, acima. Esse método está associado à variável arvore, que seria um objeto da classe ArvoreDeBusca:

```
private void btnIncluir_Click(object sender, EventArgs e)
{
    try
    {
        var umFunc = new Funcionario(int.Parse(txtChave.Text), txtNome.Text,
                                      dtpNascimento.Value, int.Parse(txtSecao.Text),
                                      int.Parse(txtMatriculaChefe.Text),
                                      int.Parse(txtDependentes.Text),
                                      float.Parse(txtSalario.Text),
                                      chkAfastado.Checked);
        arvore.IncluirNovoRegistro(umFunc);
        pnlArvore.Invalidate();
    }
    catch (Exception mens)
    {
        MessageBox.Show(mens.Message);
    }
}
```

Se o arquivo do qual lermos os dados estiver ordenado, então a árvore gerada terá ramos apenas para uma direção. Por exemplo, se as chaves dos dados fossem 87204, 87250, 88305, 89102, 90023, 91087, teríamos uma árvore em que a Raiz teria chave 87204 e todos os demais valores estariam à esquerda da árvore, formando uma lista ligada. A colocação dos novos registros seria bastante ineficiente, devido ao percurso sequencial, assim como a pesquisa, pois todos os elementos estariam em um único ramo, aumentando o tempo de pesquisa por um determinado registro. Esse tipo de árvore é chamado de árvore **degenerada**. A árvore pode ser degenerada tanto para a direita quanto para a esquerda, se os dados estiverem dispostos de tal ordem no arquivo que gere ramos apenas à direita ou à esquerda.

Essa discussão leva a um outro ponto importante. Para que a pesquisa seja eficiente, assim como a colocação de elementos, é importante que a árvore não apenas não seja degenerada, mas que também não privilegie mais um lado do que o outro. Assim, é importante que a árvore tenha ramos balanceados.

Uma árvore está balanceada quando possui, para a quantidade de nós que armazena, o menor número possível de níveis. Assim, para 20 nós, deve haver no máximo 5 níveis. Nem sempre isso é possível, mas é necessário buscar-se esse balanceamento.

Existem métodos que procuram manter a árvore balanceada dentro de certos limites, durante a modificação dinâmica dos nós (inclusões e exclusões). Estudaremos um desses métodos, as Árvores AVL.

10. Leitura balanceada do arquivo de acesso direto

Agora estudaremos como ler um arquivo de maneira a gerar uma árvore balanceada. Esse arquivo conterá dados de alunos de uma escola, sendo que cada aluno é identificado por um RA, que vemos representados na figura 7. Podemos imaginar, para a árvore dessa figura, a seguinte disposição de chaves no arquivo:

86728 85315 87230 85081 86121 86802 87291 84200 85130 86729 87150 85127 85100 85128

Essa disposição é gerada, a partir da árvore, quando a percorremos **por níveis**. Se lermos o arquivo sequencialmente, podemos montar a árvore como ela está representada na figura. O problema é que se o usuário digitar dados em ordem crescente, quando da criação do arquivo pela primeira vez, a árvore será degenerada, e o arquivo gerado pelo percurso por níveis será ordenado. Assim sendo, na próxima montagem da árvore, ela também será degenerada.

Outra abordagem procura ler os dados de maneira não sequencial, mas sim por níveis. Nessa abordagem, o arquivo deverá estar **ordenado**. Imagine que, quando **gravamos no arquivo os dados da árvore** da figura 7, o percurso para acessar os nós foi o in-ordem, de modo que os dados são gravados sequencialmente em ordem de RA, como vemos na relação abaixo:

84200	85081	85100	85127	85128	85130	85315	86121	86728	86729	86802	87150	87230	87291
0	1	2	3	4	5	6	7	8	9	10	11	12	13

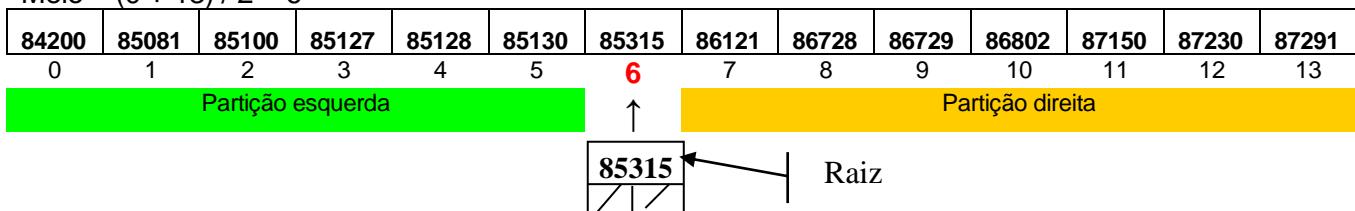
Os números abaixo dos RAs correspondem à posição de cada registro em referência ao início do arquivo.

Para montarmos a árvore, leremos o arquivo **particionadamente**, ou seja, faremos partições do mesmo, dividindo-o em partes e, a cada parte, leremos o registro do meio dessa parte e o colocaremos na árvore, num novo nó.

Particiona-se o arquivo ao **meio**, ou seja, **leremos diretamente o registro que fica na posição média do arquivo**, e os dados desse registro lido são armazenados na árvore. Como particionou-se ao meio e o arquivo está **ordenado**, após o registro lido haverá metade dos registros do arquivo com **chave maior** que a chave lida e, antes do registro lido, outra metade do arquivo, com **chave menor** do que a chave lida, estarão gravada.

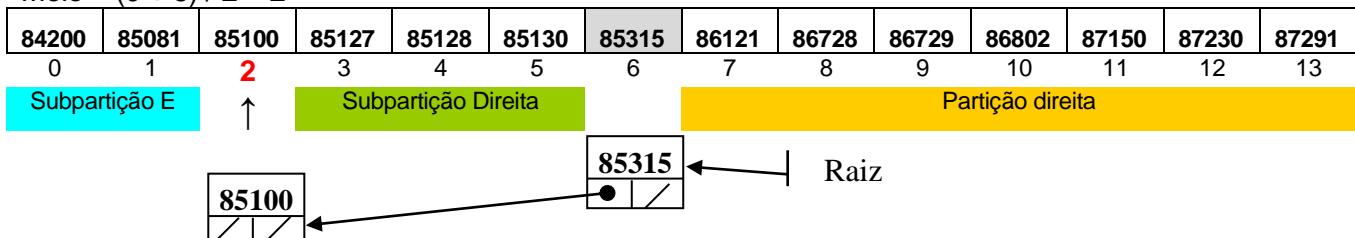
O primeiro registro lido será o primeiro registro a ser colocado na árvore e, portanto, será armazenado na **Raiz** da Árvore:

$$\text{Meio} = (0 + 13) / 2 = 6$$



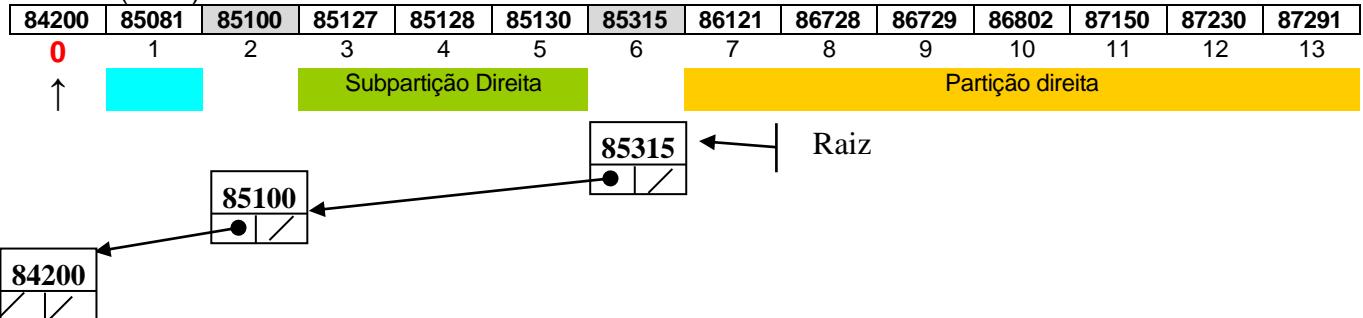
Os registros com chave **menor** que a lida, que formam metade do arquivo, serão armazenados no **ramo esquerdo** e a outra metade do arquivo será armazenada no ramo **direito** (com chave **maior**) a partir da raiz. Vamos agora particionar a partição esquerda, dividindo-a pela metade; leremos o registro da posição média dessa partição e o armazenaremos na árvore, como vemos na figura abaixo:

$$\text{Meio} = (0 + 5) / 2 = 2$$



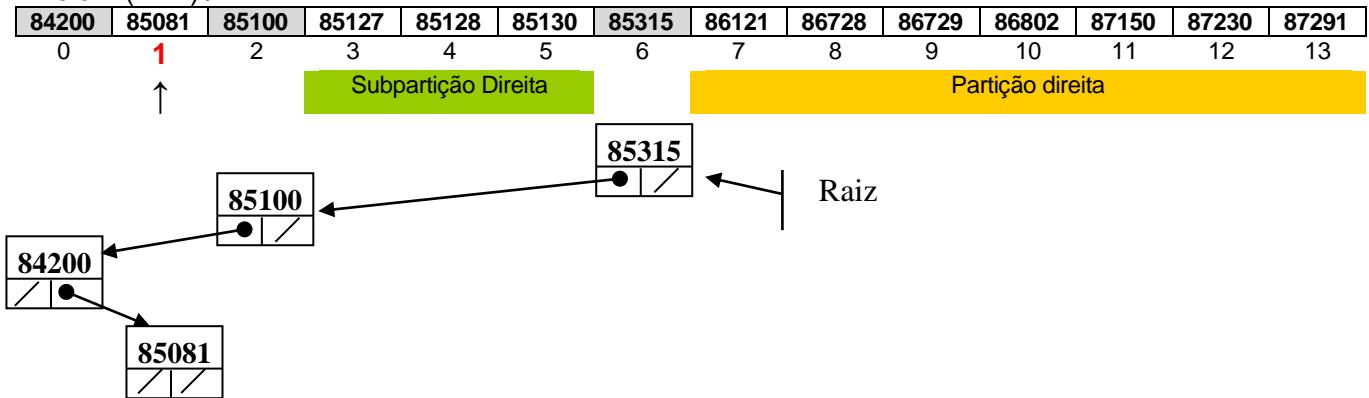
Novamente particionamos a metade esquerda da partição atual, dividindo-a pela metade; leremos o registro da posição média dessa partição e o armazenaremos na árvore, como vemos na figura abaixo:

$$\text{Meio} = (0 + 1) / 2 = 0$$



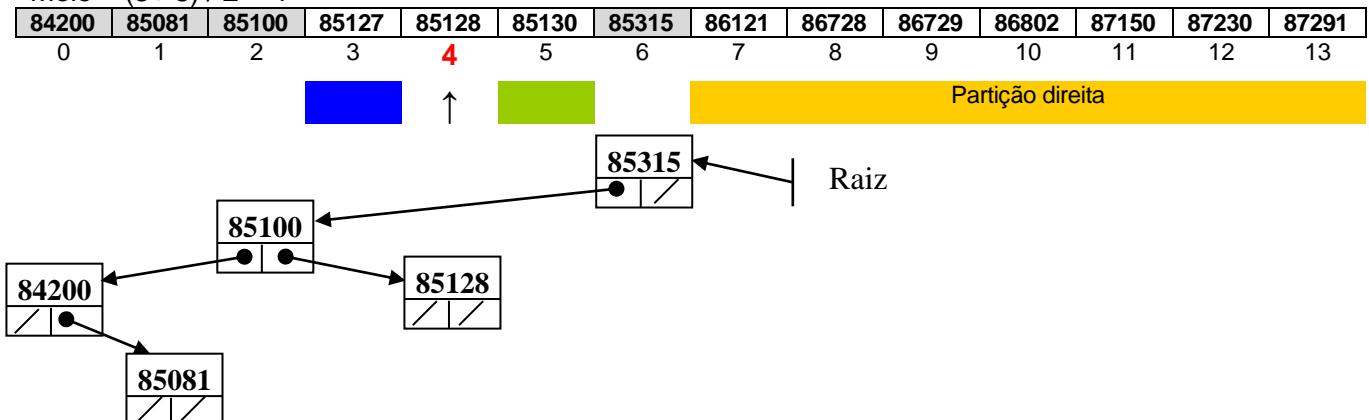
Se tentarmos novamente dividir a metade esquerda da partição atual, veremos que o índice do **início ficou maior que o** índice do **fim dessa partição** ($0 > -1$) e, assim, não temos mais onde dividir. Nesse momento, o fluxo de execução **recursivo** retorna ao local de chamada e passa a dividir a metade direita da partição atual, como vemos abaixo:

$$\text{Meio} = (1+ 1) / 2 = 1$$



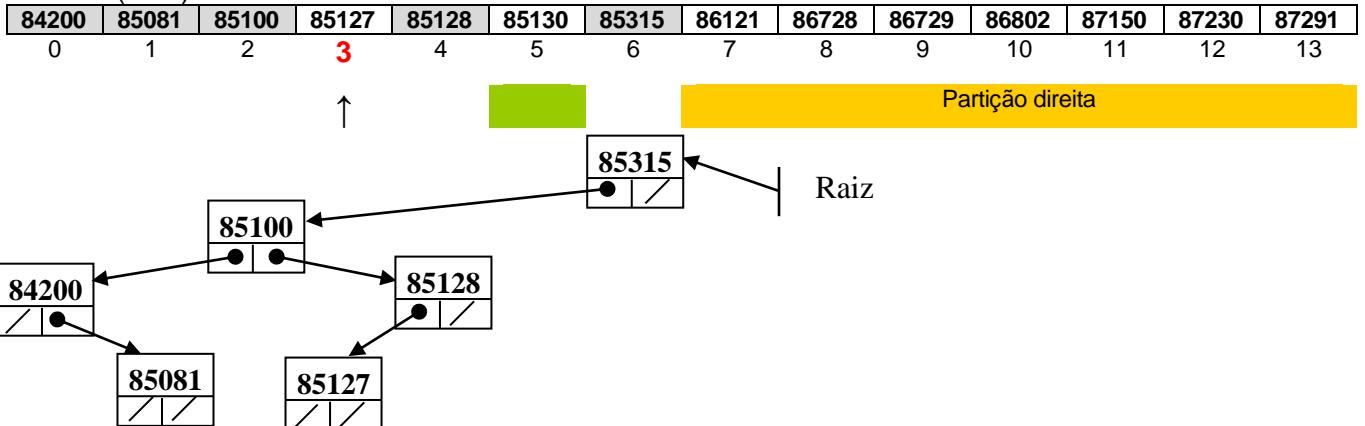
Tentamos partitionar a metade esquerda da partição atual e, novamente, o índice de início da mesma será maior que o de fim. Assim, não há o que partitionar e retornamos ao local de chamada anterior, partitionando a metade direita da primeira partição esquerda (em cor verde clara):

$$\text{Meio} = (3+ 5) / 2 = 4$$



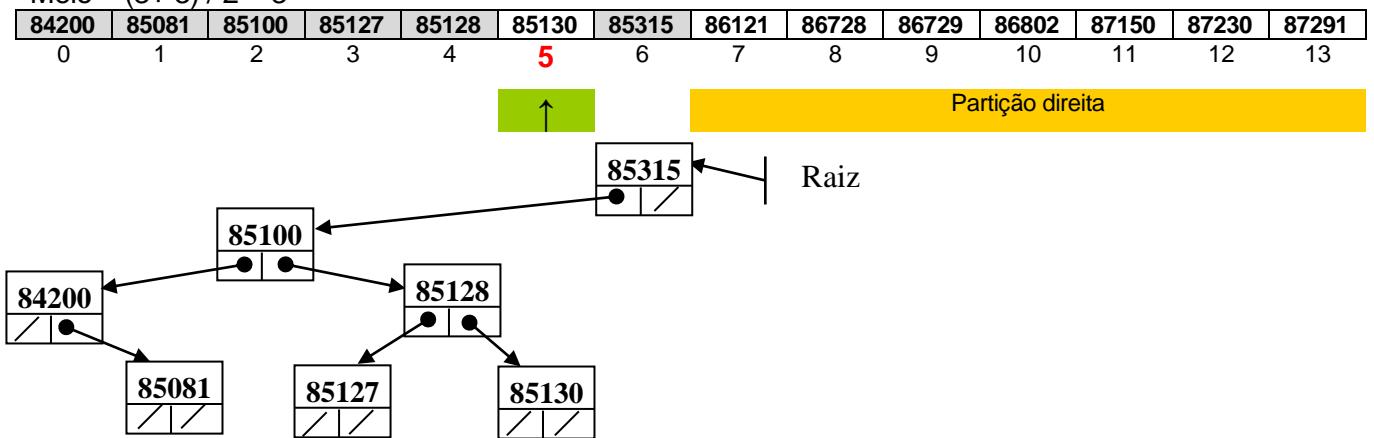
Agora particionamos a metade esquerda da partição atual (**em azul**).

$$\text{Meio} = (3+3) / 2 = 3$$



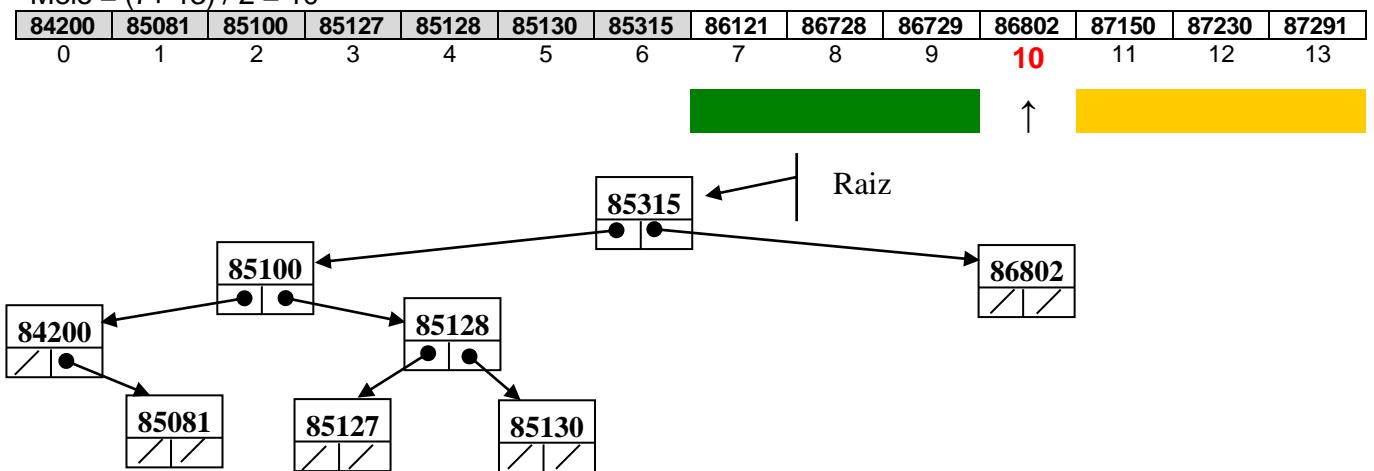
Após isso, tentamos particionar a metade esquerda da partição atual e veremos que o índice de início ficou maior que seu índice de fim. Tentamos particionar a metade direita e vemos que o índice de fim ficou menor que o índice de início. Assim, retornamos ao local de chamada anterior e passamos a particionar a metade direita da partição à qual retornamos, e que está pintada **de verde**.

$$\text{Meio} = (3+3) / 2 = 3$$



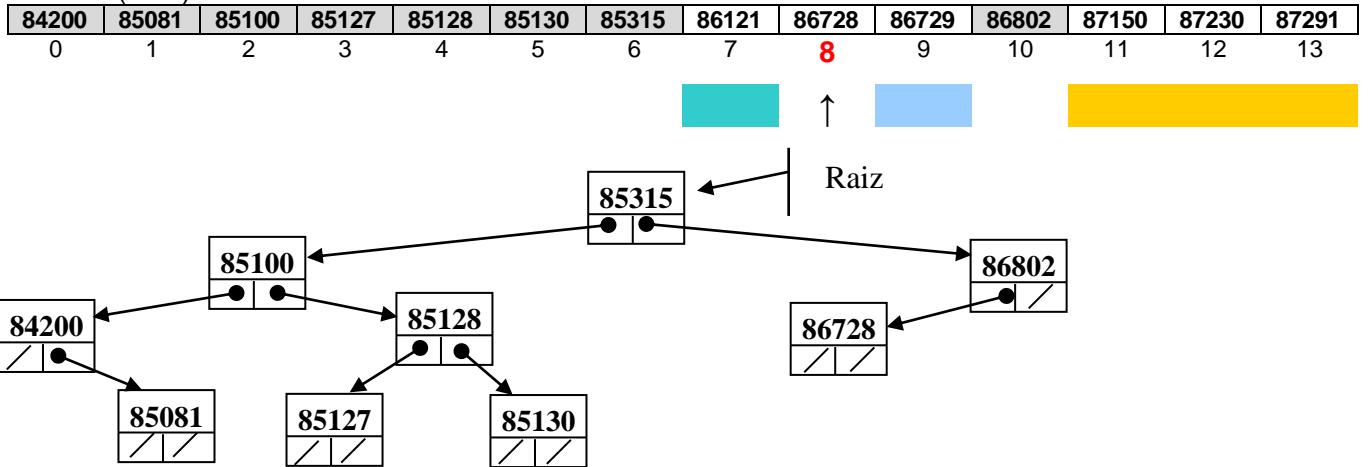
Como não há mais o que particionar, tanto à esquerda quanto à direita do registro atual, retornamos ao local de chamada e iremos à direita da partição atual (que na verdade é a primeira de todas) e particionaremos a região pintada **de laranja**.

$$\text{Meio} = (7+13) / 2 = 10$$



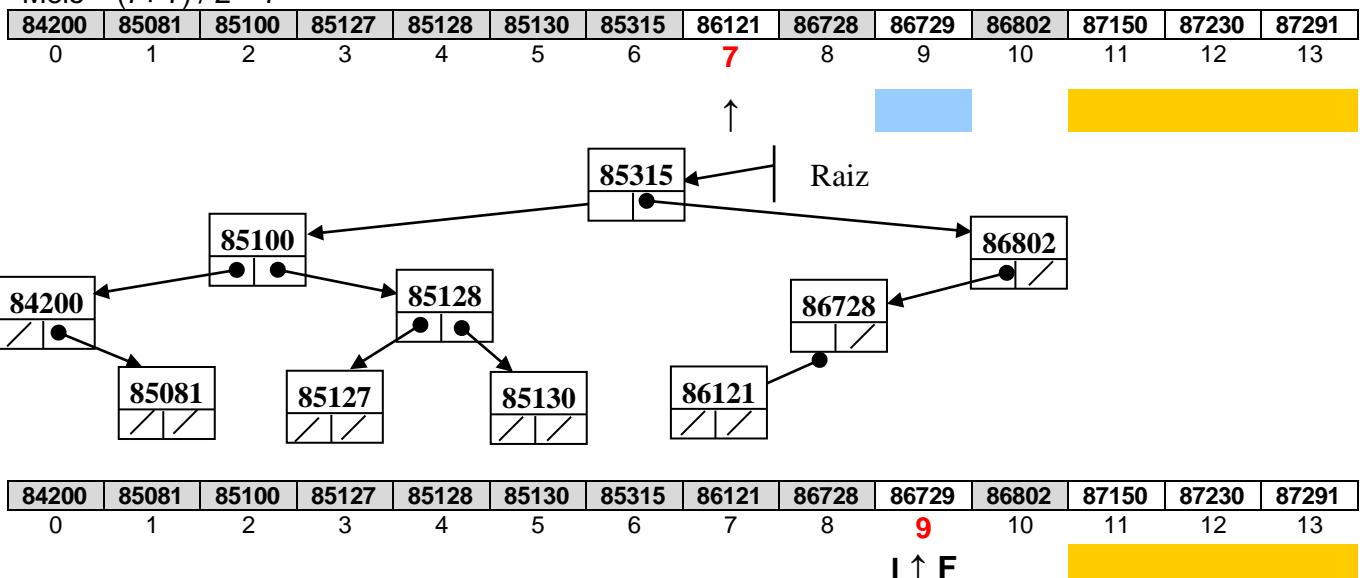
Agora, vamos partitionar a metade esquerda da partição atual, que estava na cor de verde escuro:

$$\text{Meio} = (7+9)/2 = 8$$

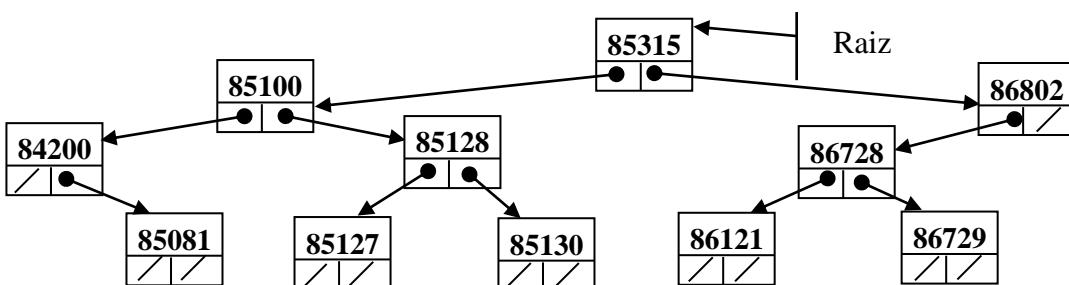


Prosseguindo com esse método, progressivamente acessaremos cada registro e o colocaremos na árvore, na posição correta, como vemos na sequência de figuras abaixo:

$$\text{Meio} = (7+7)/2 = 7$$

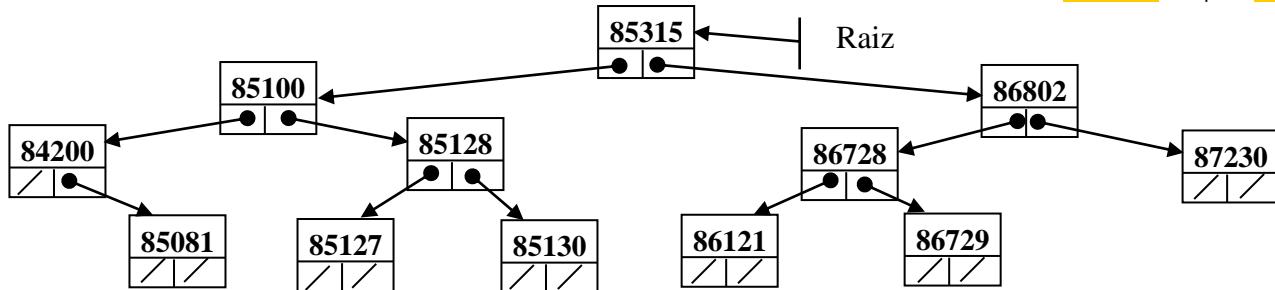


I ↑ F



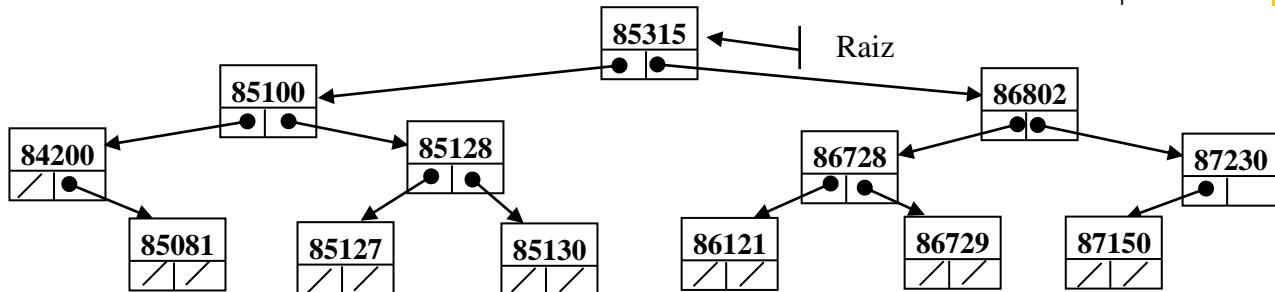
84200	85081	85100	85127	85128	85130	85315	86121	86728	86729	86802	87150	87230	87291
0	1	2	3	4	5	6	7	8	9	10	11	12	13

I ↑ F



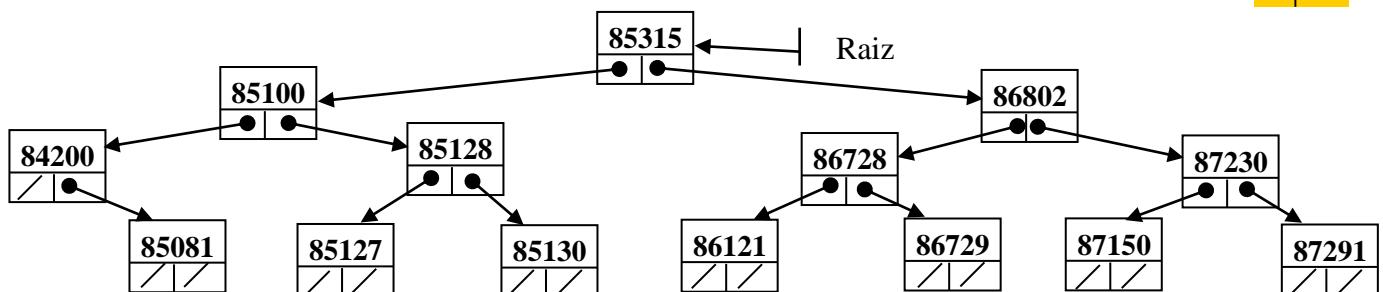
84200	85081	85100	85127	85128	85130	85315	86121	86728	86729	86802	87150	87230	87291
0	1	2	3	4	5	6	7	8	9	10	11	12	13

I ↑ F



84200	85081	85100	85127	85128	85130	85315	86121	86728	86729	86802	87150	87230	87291
0	1	2	3	4	5	6	7	8	9	10	11	12	13

I ↑ F



Ao final do processo acima, teríamos todo o arquivo lido e a árvore montada, como vemos na figura abaixo. Observe que a árvore está balanceada, ao contrário da árvore ilustrada na figura 7. Portanto, esse método de ler um arquivo particionando suas metades leva ao balanceamento de uma árvore logo em sua leitura o que, por sua vez, terá impactos benéficos na velocidade de pesquisa nessa árvore.

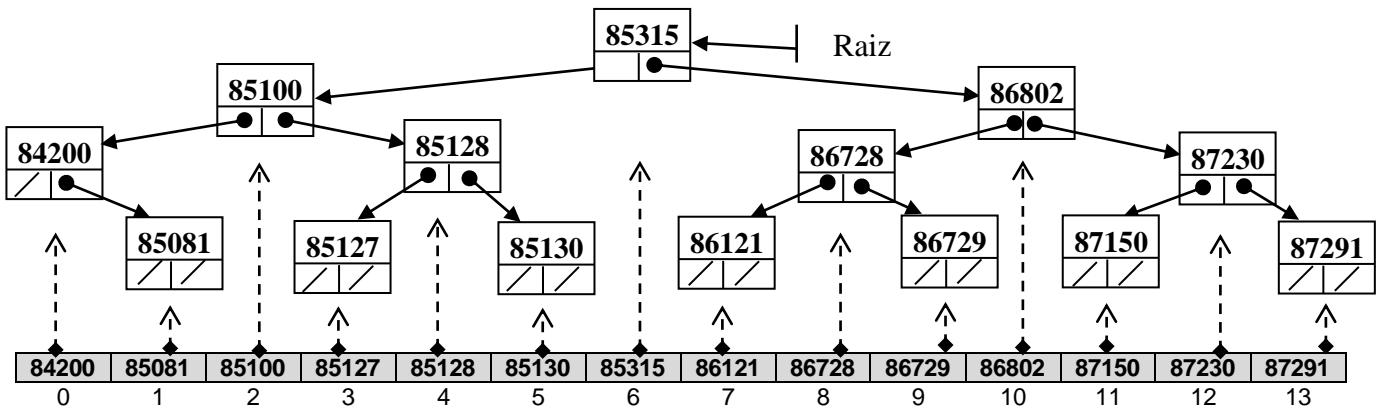


Figura 9– Árvore binária平衡ada e os registros que deram origem aos nós mediante particionamento de arquivo

O arquivo com os dados a serem lidos deverá ser de acesso aleatório, para que possamos lê-fora da ordem sequencial, usando o método seek para posicionar no registro a ser lido. Poderemos usar o método recursivo Particionar(), codificado abaixo, para ler os dados na ordem em que eles devem ser colocados na árvore, que ficaria balanceada logo de início.

```
public void LerArquivoDeRegistros(string nomeArquivo)
{
    raiz = null;
    Dado dado = new Dado();
    var origem = new FileStream(nomeArquivo, FileMode.OpenOrCreate);
    var arquivo = new BinaryReader(origem);
    int posicaoFinal = (int)origem.Length / dado.TamanhoRegistro - 1;
    Particionar(0, posicaoFinal, ref raiz);
    origem.Close();

    void Particionar(long inicio, long fim, ref NoArvore<Dados> atual)
    {
        if (inicio <= fim)
        {
            long meio = (inicio + fim) / 2;
            dado = new Dado(); // cria um objeto para armazenar os dados
            dado.LerRegistro(arquivo, meio); //
            atual = new NoArvore<Dados>(dado);
            var novoEsq = atual.Esq;
            Particionar(inicio, meio - 1, ref novoEsq); // Particiona à esquerda
            atual.Esq = novoEsq;
            var novoDir = atual.Dir;
            Particionar(meio + 1, fim, ref novoDir); // Particiona à direita
            atual.Dir = novoDir;
        }
    }
}
```

reflita sobre a importância desta passagem por referência na ligação dos nós da árvore

Observe que a inclusão foi feita diretamente, sem a chamada ao método recursivo Incluir() da classe Arvore. Isso é possível devido à passagem por referência que C# implementa. Em Java não existe passagem de parâmetros por referência, de modo que teríamos de chamar o método de inclusão.

11. Exclusão de Registros de uma árvore de busca

O grande problema de se retirar um registro de uma árvore de busca é que deve-se mantê-la com as características de ordenação que fazem dela uma árvore de busca. O processo de exclusão deve levar

em consideração diversas possibilidades, para não perdermos informações nem descharacterizar a ordenação da árvore. Há três casos que se deve solucionar:

1. O nó a ser excluído é uma folha
2. O nó a ser excluído tem um único filho
3. O nó a ser excluído tem dois filhos

Para ser excluído, um elemento deve ser procurado na árvore binária e possuir um apontador que o refencie. Usaremos a figura abaixo para ilustrar nosso estudo:

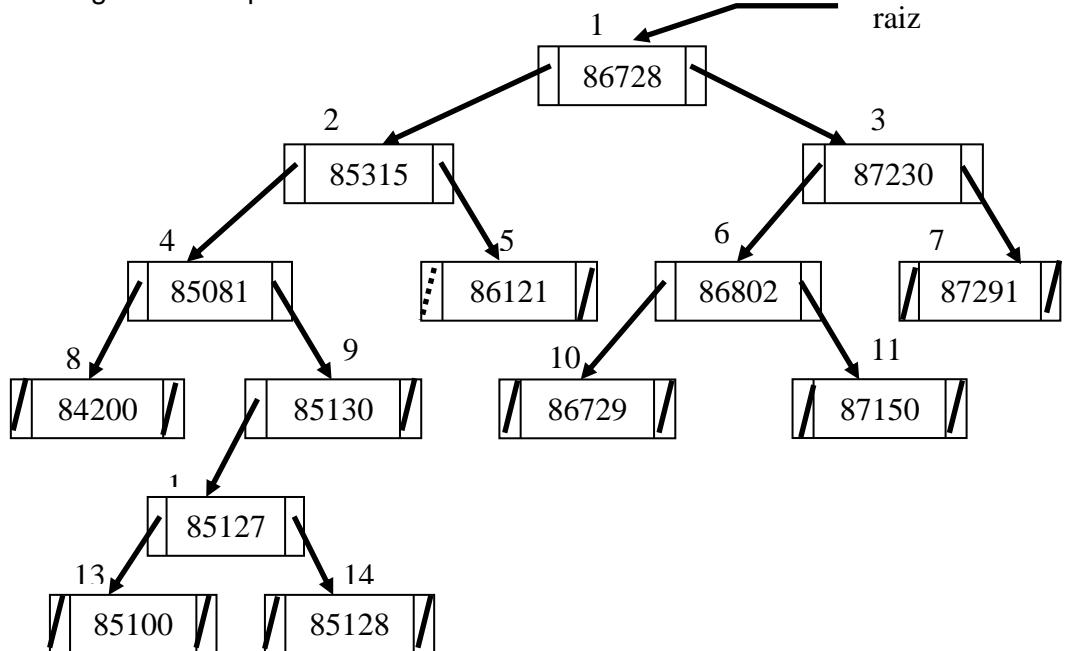


Figura 10 – Árvore binária original para estudo do processo de exclusão de nós mantendo a árvore ordenada

1. Digamos que desejamos retirar o elemento com RA 86729. Como este elemento é uma folha, basta apenas que a ligação com seu antecessor (86802) seja interrompida (passe a valer null). Assim, 86802 passa a ter como descendente apenas o nó à sua direita (87150).

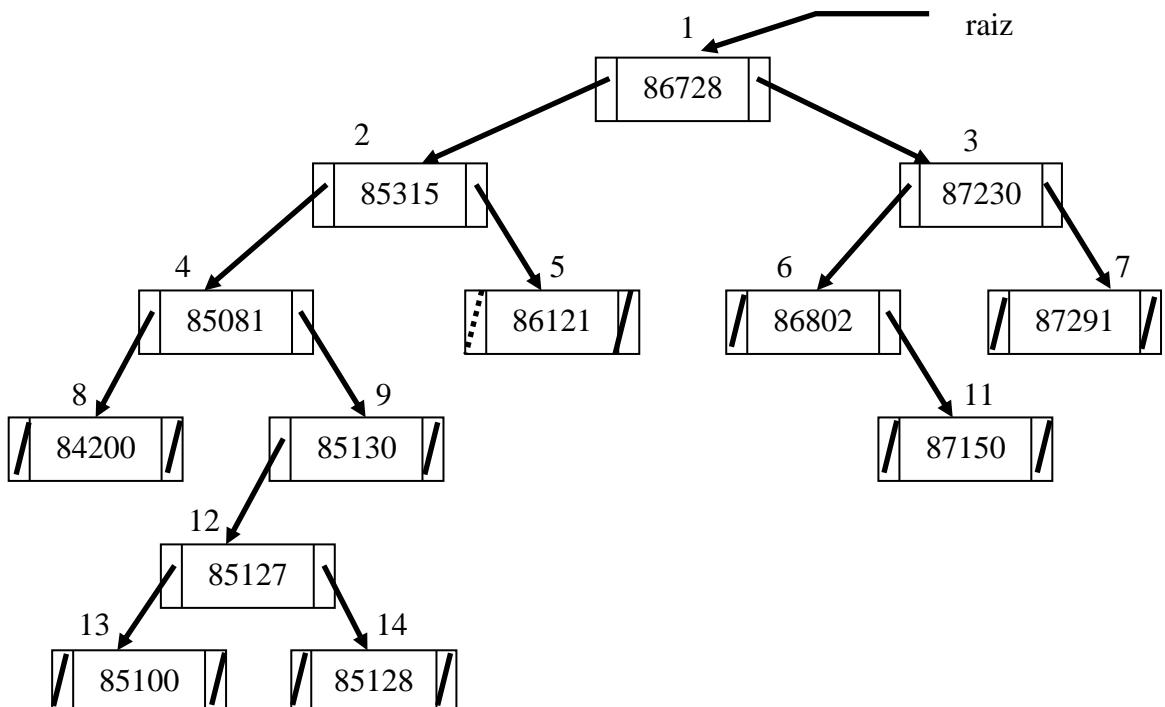


Figura 11– Exclusão de um nó folha, que não possui descendentes (antigo nó 10, com chave 86729)

2. Caso se deseje excluir o elemento com RA 85130 (nó 9), deve-se fazer com que seu antecessor (85081) aponte para o único descendente de 85130 (em outras palavras, a sub-árvore do nó 12 passa a ser descendente direta do nó 4). Como 85127 é maior que 85081, e todos os seus descendentes também o serão (já que estão à direita de 85081), a árvore continua ordenada, após as duas operações efetuadas:

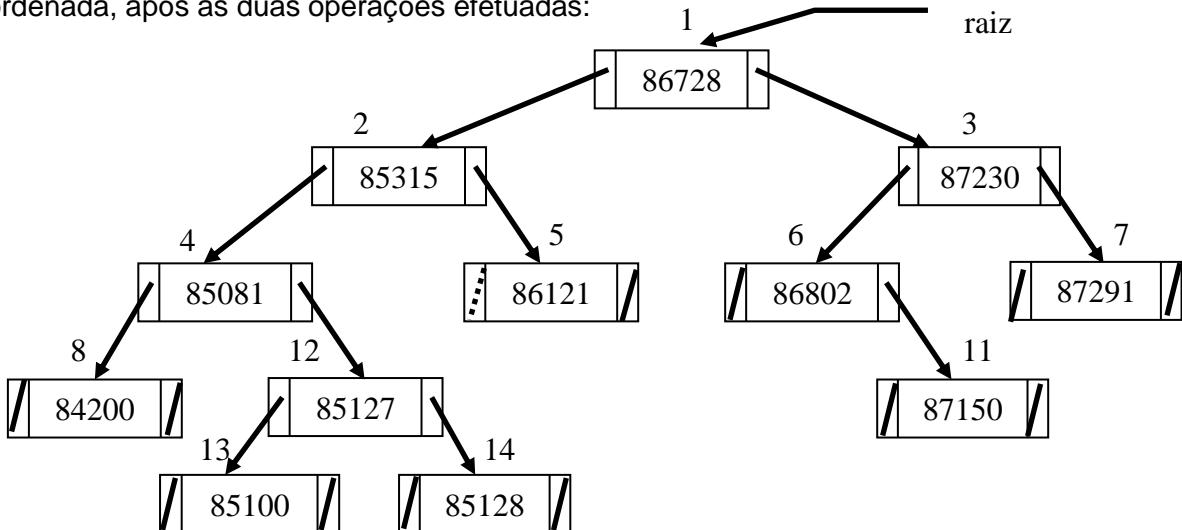


Figura 12 – Exclusão de um nó com um único descendente (antigo nó 9, com chave 85130)

3. Agora, caso se deseje excluir o elemento com RA 86728, que possui os dois descendentes, a árvore deverá sofrer uma operação de reorganização, para que se encontre um nó que possa substituí-lo. Note que esse caso ocorreria também para os elementos 2, 3, 4 e 12 da figura anterior.

Para efetuar a exclusão do elemento 86728, que possui dois descendentes, deve-se encontrar o maior elemento da sua sub-árvore esquerda (ou seja, o maior dos menores). Descemos para o nó logo à esquerda e, em seguida, seguimos sempre para a direita até não podermos mais. Esse elemento será uma folha, ou então um elemento sem descendente direito (se tivesse, esse descendente, por estar à direita, seria maior que o antecessor). No nosso caso, é o elemento com RA 86121, ou seja, o nó 5. A partir daí, troca-se este nó com o nó que queremos retirar. Se colocarmos a chave 86121 no lugar da chave 86728, a organização ordenada da árvore não será afetada.

Deve-se também atualizar o nó que aponta para a nova raiz da sub-árvore. No caso de o nó excluído ser a raiz da sub-árvore, o apontador de raiz passa a apontar para o descendente esquerdo do nó retirado. No caso de não ser a raiz, deve-se fazer com que o antecessor do nó retirado aponte para o descendente esquerdo do nó retirado. Assim, chegamos à árvore abaixo:

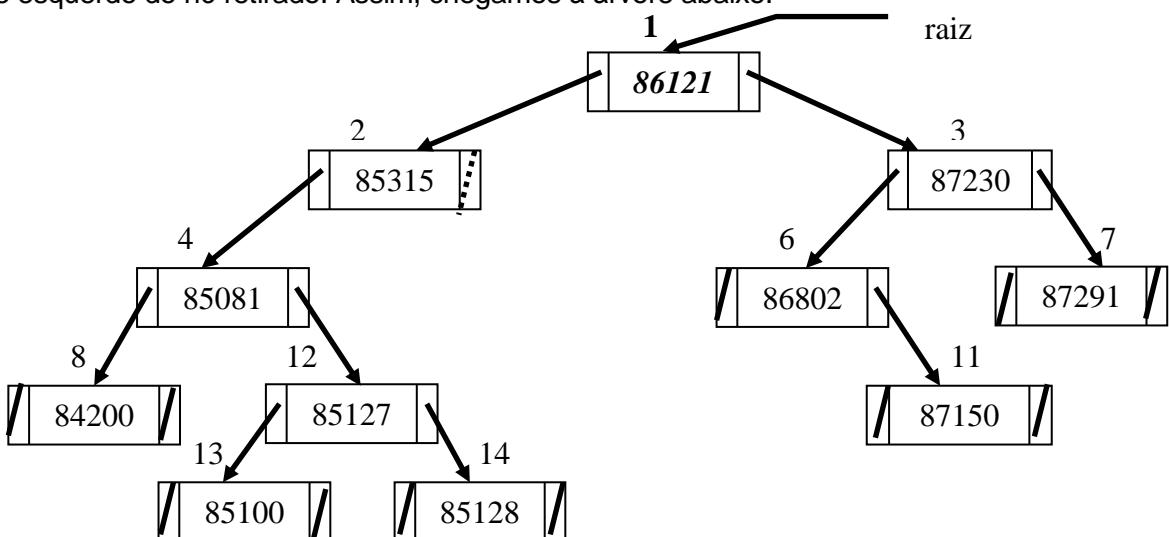


Figura 13 – Exclusão de um nó com dois descendentes (antigo nó 1, com chave 86728)

Note que o campo Direita do nó 2 recebeu o valor do campo Esquerda do antigo nó 5 (**null**). Como esse valor era **null**, 2^.Direita valerá **null**. Se o nó 5 tivesse descendentes à esquerda, estes seriam apontados pelo nó 2. Além disso, os valores que estavam no nó 5 foram copiados para o nó 1 (o nó que continha a chave a excluir). Essa artimanha evita que se tenha de deslocar fisicamente os nós para que 86121 passe a ser a raiz da árvore (ou da sub-árvore) reorganizada pela exclusão. O nó que será excluído fisicamente (com o Garbage Collector) será o nó 5.

Abaixo, temos o algoritmo que implementa esse processo:

```
public bool Excluir(Dado procurado)
{
    return Excluir(ref raiz);

bool Excluir(ref NoArvore<Dado> atual)
{
    NoArvore<Dado> atualAnt;
    if (atual == null)
        return false;
    else
        if (atual.Info.CompareTo(procurado) > 0)
        {
            var temp = atual.Esq;
            bool result = Excluir(ref temp);
            atual.Esq = temp;
            return result;
        }
        else
            if (atual.Info.CompareTo(procurado) < 0)
            {
                var temp = atual.Dir;
                bool result = Excluir(ref temp);
                atual.Esq = temp;
                return result;
            }
        else
            {
                atualAnt = atual; // nó a retirar
                if (atual.Dir == null)
                    atual = atual.Esq;
                else
                    if (atual.Esq == null)
                        atual = atual.Dir;
                    else
                        { // pai de 2 filhos
                            var temp = atual.Esq;
                            Rearranjar(ref temp, ref atualAnt);
                            atual.Esq = temp;
                            atualAnt = null; // libera o nó excluído
                        }
                return true;
            }
}
```

```
void Rearranjar(ref NoArvore<Dado> aux, ref NoArvore<Dado> atualAnt)
{
    if (aux.Dir != null)
    {
        NoArvore<Dado> temp = aux.Dir;
        Rearranjar(ref temp, ref atualAnt); // Procura Maior
```

```
        aux.Dir = temp;
    }
    else
    {
        // Guarda os dados do nó a excluir
        atualAnt.Info = aux.Info; // troca conteúdo!
        atualAnt = aux;          // funciona com a passagem por referência
        aux = aux.Esq;
    }
}
```

Uma outra versão da remoção de nós – não recursiva (adaptado do autor Michael McMillan)

As operações que vimos até agora na árvore não foram muito complicadas, ao menos em comparação com a exclusão de nós, que revisaremos agora. Em alguns casos, remover um nó da árvore é quase trivial; em outros casos, é bastante complicado e exige que tomemos especial cuidado no código, pois de outra forma correremos o risco de destruir a ordem hierárquica correta da árvore binária de busca.

Começaremos examinando a remoção de um nó da árvore pelo caso mais simples – remoção de uma folha.

A remoção de uma folha é o caso mais simples porque não há nós descendentes (filhos) para considerar. Tudo o que temos a fazer é colocar null no ponteiro do nó **antecessor** que faz a ligação com o nó que queremos remover. Obviamente o nó ainda estará na memória, mas como ele não terá mais referências a ele, logo será retirado da memória pelo mecanismo de coleta de lixo.

O fragmento de código a seguir efetua a remoção de um nó folha, constando também do início do método ApagarNo, que declara alguns dados internos e posiciona um apontador no nó a ser apagado:

```
public bool ApagarNo(Tipo chaveARemover) // retorna true se achou e removeu o nó procurado
{
    atual = raiz;
    antecessor = null;
    bool ehFilhoEsquerdo = true; // enquanto não achou chave a remover
    while (atual != null && atual.Info.CompareTo(chaveARemover) != 0)
    {
        antecessor = atual; // põe uma referência ao nó "pai" daquele que será excluído
        if (atual.Info.CompareTo(chaveARemover) > 0)
        {
            ehFilhoEsquerdo = true;
            atual = atual.Esq;
        }
        else
        {
            ehFilhoEsquerdo = false;
            atual = atual.Dir;
        }
    } // fim do while
    if (atual == null) // neste caso, a chave a remover não existe e não pode
        return false; // ser excluída, dai retornamos falso indicando isso
}
```

O while nos leva ao nó que desejamos apagar. Caso o nó **não** tenha sido encontrado, o **return** acima providencia que o fluxo de execução saia deste método, devolvendo **false** como resultado. Em seguida, caso tenhamos encontrado o nó que se deseja excluir, a variável **atual** o referencia e a variável **antecessor** referencia o nó que se liga a **atual**. Agora temos que verificar em que situação **atual** se encontra na árvore:

```
// se o fluxo de execução vem para este ponto, a chave a remover foi encontrada
// e o ponteiro atual indica o nó que contém essa chave

if ((atual.Esq == null) && (atual.Dir == null)) // é folha, nó com 0 filhos
{
    if (atual == raiz)
        raiz = null; // exclui a raiz e a árvore fica vazia
    else
        if (ehFilhoEsquerdo) // se for filho esquerdo, o antecessor deixará
            antecessor.Esq = null; // de ter um descendente esquerdo
        else // se for filho direito, o antecessor deixará de
            antecessor.Dir = null; // apontar para esse filho

    atual = antecessor; // feito para atual apontar um nó válido ao sairmos do método
}
else // verificará as duas outras possibilidades, exclusão de nó com 1 ou 2 filhos
...
// o restante do método virá a partir daqui
```

O primeiro teste verifica se o filho esquerdo e o filho direito daquele nó são ambos nulos. Se forem, testamos se o nó a ser excluído é a raiz (nesse caso, a árvore tem apenas um nó, o nó raiz, e portanto ela ficará vazia ao excluirmos esse nó). Caso o nó a ser excluído não seja a raiz, fazemos com que o antecessor aponte nulo na ligação que tinha ao nó excluído.

Removendo um nó com um único descendente

Quando o nó a ser removido tem um único descendente, existem quatro condições a verificar:

1. O descendente do nó a excluir pode ser um descendente esquerdo;
2. O descendente do nó a excluir pode ser um descendente direito;
3. O nó a ser removido pode ser um descendente esquerdo do seu antecessor;
4. O nó a ser removido pode ser um descendente direito do seu antecessor.

Eis o fragmento de código:

```
if ((atual.Esq == null) && (atual.Dir == null)) // é folha
{
    if (atual == raiz)
        raiz = null;
    else
        if (ehFilhoEsquerdo)
            antecessor.Esq = null;
        else
            antecessor.Dir = null;

    atual = antecessor; // feito para atual apontar um nó válido ao fim do método
}
else // começamos a escrever a partir deste else
    if (atual.Dir == null) // neste caso, só tem o filho esquerdo
    {
        if (atual == raiz)
            raiz = atual.Esq;
        else
            if (ehFilhoEsquerdo)
                antecessor.Esq = atual.Esq;
            else
                antecessor.Dir = atual.Esq;
        atual = antecessor;
    }
else
```

Já escrito anteriormente

```

if (atual.Esq == null) // neste caso, só tem o filho direito
{
    if (atual == raiz)
        raiz = atual.Dir;
    else
        if (ehFilhoEsquerdo)
            antecessor.Esq = atual.Dir;
        else
            antecessor.Dir = atual.Dir;
        atual = antecessor;
}
else
    ... // neste caso, tem os dois filhos, esquerdo e direito
    // o restante do método vem aqui
}

```

Primeiramente, testamos se o nó direito é nulo. Se for, então testamos se estamos na raiz. Se estivermos, movemos o descendente esquerdo para o nó raiz. Por outro lado, se o nó é um descendente esquerdo configuramos o nó esquerdo do novo antecessor para se tornar o nó esquerdo atual, os se temos um descendente direito, o nó direito do antecessor passará a ser o nó direito do nó atual.

Removendo um nó com dois descendentes

A exclusão de nós se torna mais complicada quando temos de excluir um nó com dois descendentes.

Observe a figura ao lado. Se precisamos excluir o nó com a chave 52, teremos de reconstruir a árvore. Não podemos substituir o nó 52 com o nó 54, pois 52 possui dois filhos e, assim, seu antecessor (45), não poderá apontar para esses dois descendentes com apenas o seu ponteiro à direita.

A resposta a esse problema é mover o sucessor in-ordem no lugar do nó excluído. Esse sucessor é o nó que, num percurso in-ordem vem logo depois do nó a ser excluído (no caso, 54 vem logo após 52 no percurso in-ordem).

Essa solução funciona corretamente **a menos que** o sucessor também tenha um descendente esquerdo (no caso da figura, o 53), mas também existe uma forma de corrigir esse problema.

A figura ao lado ilustra como o sucessor in-ordem funciona. A idéia é, em resumo, trocar o conteúdo do nó a ser excluído com o conteúdo do nó que contém a chave que seja a **menor dos maiores sucessores** da chave a ser excluída. Em outras palavras, trocamos o conteúdo do nó que queremos excluir pelo **conteúdo** do nó que é o **menor dos seus maiores descendentes**.

Para encontrar esse descendente, posicionamos o percurso no filho direito do nó a ser excluído. Assim, na árvore da figura, apontamos o nó 54,

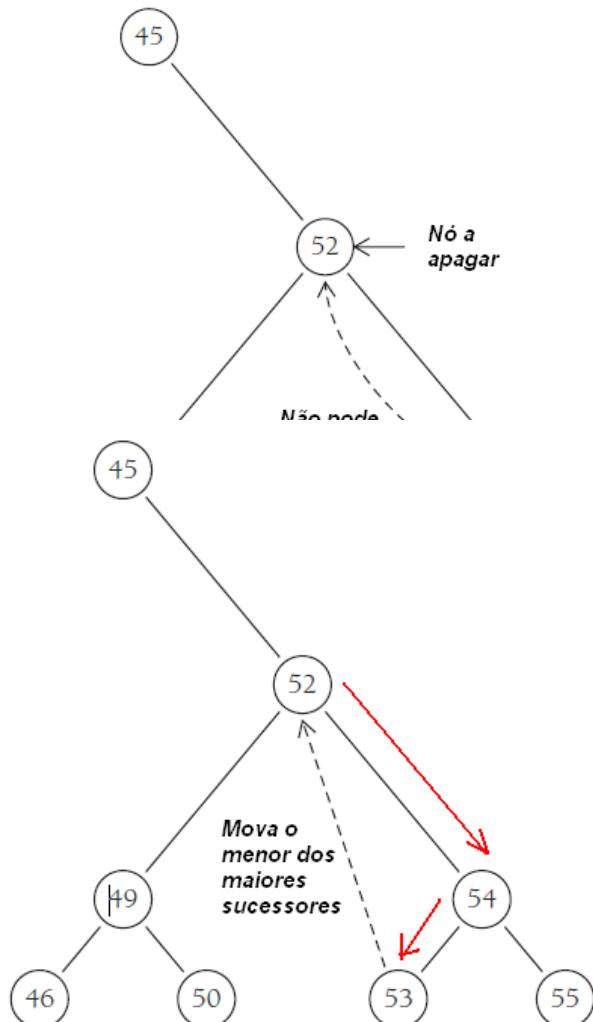


Figura 15 – Movendo o sucessor in-ordem

que é o filho direito do nó a ser excluído. Nesse ramo, teremos apenas chaves maiores que a chave que desejamos excluir, pois a árvore é de busca, o que a torna ordenada.

Após isso, começamos a percorrer todos os descendentes esquerdos desse nó, até que não haja mais descendentes. Já que o menor valor de uma sub-árvore deve estar no final do caminho dado pelos filhos esquerdos da raiz da sub-árvore, seguindo o caminho à esquerda nos deixará no menor nó que é maior do que a raiz da sub-árvore percorrida (ou seja, nesta discussão, o nó que desejamos excluir, com a chave 52).

Aqui temos o código para encontrar o sucessor de um nó excluído:

```
public NoArvore<Tipo> ProcuraMenorDosMaioresDescendentes (NoArvore<Tipo> noAExcluir)
{
    NoArvore<Tipo> paiDoSucessor=noAExcluir;
    NoArvore<Tipo> sucessor = noAExcluir;
    NoArvore<Tipo> atualExc = noAExcluir.Dir;
    while (atualExc != null)
    {
        if (atualExc.Esq != null)
            paiDoSucessor = sucessor;
        sucessor = atualExc;
        atualExc = atualExc.Esq;
    }
    if (sucessor != NoParaExcluir.Dir)
        paiDoSucessor.Esq = sucessor.Dir;
    return sucessor;
}
```

O código abaixo faz a ligação faz o ramo esquerdo do antecessor do sucessor apontar para o ramo direito do sucessor (que, se tiver descendentes, serão à direita)

```
if (sucessor != noAExcluir.Dir)
    paiDoSucessor.Esq = sucessor.Dir;
return sucessor;
```

O código final no método ApagarNo é :

```
else // tem os dois descendentes
{
    NoArvore<Dados> menorDosMaiores = ProcuraMenorDosMaioresDescendentes(atual);
    atual.Info = menorDosMaiores.Info; // substitui dados armazenados
    menorDosMaiores = null; // para liberar o nó trocado da memória
}
```

Isto completa o código para o método ApagarNo. Por este método ser relativamente complicado, algumas implementações de árvores simplesmente marcam nós para remoção e incluem código para verificar esta marca quando realizando pesquisas e percursos.

Segue o código completo do método ApagarNo():

```
public bool ApagarNo(Dado registroARemover)
{
    atual = raiz;
    antecessor = null;
    bool ehFilhoEsquerdo = true;
    while (atual.Info.CompareTo(registroARemover) != 0) // enquanto não acha a chave a remover
    {
        antecessor = atual;
        if (atual.Info.CompareTo(registroARemover) > 0)
```

```
{  
    ehFilhoEsquerdo = true;  
    atual = atual.Esq;  
}  
else  
{  
    ehFilhoEsquerdo = false;  
    atual = atual.Dir;  
}  
  
if (atual == null) // neste caso, a chave a remover não existe e não pode  
    return false; // ser excluída, dai retornamos falso indicando isso  
} // fim do while  
  
// se fluxo de execução vem para este ponto, a chave a remover foi encontrada  
// e o ponteiro atual indica o nó que contém essa chave  
if ((atual.Esq == null) && (atual.Dir == null)) // é folha, nó com 0 filhos  
{  
    if (atual == raiz)  
        raiz = null; // exclui a raiz e a árvore fica vazia  
    else  
        if (ehFilhoEsquerdo) // se for filho esquerdo, o antecessor deixará  
            antecessor.Esq = null; // de ter um descendente esquerdo  
        else // se for filho direito, o antecessor deixará de  
            antecessor.Dir = null; // apontar para esse filho  
  
    atual = antecessor; // feito para atual apontar um nó válido ao sairmos do método  
}  
else // verificará as duas outras possibilidades, exclusão de nó com 1 ou 2 filhos  
if (atual.Dir == null) // neste caso, só tem o filho esquerdo  
{  
    if (atual == raiz)  
        raiz = atual.Esq;  
    else  
        if (ehFilhoEsquerdo)  
            antecessor.Esq = atual.Esq;  
        else  
            antecessor.Dir = atual.Esq;  
    atual = antecessor;  
}  
else  
if (atual.Esq == null) // neste caso, só tem o filho direito  
{  
    if (atual == raiz)  
        raiz = atual.Dir;  
    else  
        if (ehFilhoEsquerdo)  
            antecessor.Esq = atual.Dir;  
        else  
            antecessor.Dir = atual.Dir;  
    atual = antecessor;  
}  
else // tem os dois descendentes  
{  
    NoArvore<Dados> menorDosMaiores = ProcuraMenorDosMaioresDescendentes(atual);  
    atual.Info = menorDosMaiores.Info;  
    menorDosMaiores = null; // para liberar o nó trocado da memória  
}  
return true;  
}
```

```
public NoArvore<Dado> ProcuraMenorDosMaioresDescendentes(NoArvore<Dado> noAExcluir)
{
    NoArvore<Dado> paiDoSucessor = noAExcluir;
    NoArvore<Dado> sucessor = noAExcluir;
    NoArvore<Dado> atual = noAExcluir.Dir;    // vai ao ramo direito do nó a ser excluído,
                                                // pois este ramo contém os descendentes que
                                                // são maiores que o nó a ser excluído
    while (atual != null)
    {
        if (atual.Esq != null)
            paiDoSucessor = atual;
        sucessor = atual;
        atual = atual.Esq;
    }
    if (sucessor != noAExcluir.Dir)
    {
        paiDoSucessor.Esq = sucessor.Dir;
        sucessor.Dir = noAExcluir.Dir;
    }
    return sucessor;
}
```

12. Salvamento de Árvores de Busca em arquivos

Para salvar os dados de uma árvore em um arquivo, basta percorrer a árvore, de qualquer forma. No entanto, no momento de ler esses dados para novamente armazená-los na memória, em uma árvore de busca, a forma como os dados estão dispostos afetará o seu balanceamento.

Portanto, a maneira mais eficiente de salvar os dados e, ao relê-los, facilitar o balanceamento da árvore usando o método do particionamento que estudamos anteriormente, é percorrer a árvore com o percurso in-ordem, de modo que as chaves fiquem ordenadas crescentemente e, quando do particionamento, leia-se primeiramente o registro correspondente à raiz da árvore (o registro do meio do arquivo), e se siga o particionamento como estudamos.

Abaixo temos o método da classe Arvore que implementa essa funcionalidade:

```
public void GravarArquivoDeRegistros(string nomeArquivo)
{
    var destino = new FileStream(nomeArquivo, FileMode.Create);
    var arquivo = new BinaryWriter(destino);
    GravarInOrdem(raiz);
    arquivo.Close();

    void GravarInOrdem(NoArvore<Dado> r)
    {
        if (r != null)
        {
            GravarInOrdem(r.Esq);
            r.Info.GravarRegistro(arquivo);
            GravarInOrdem(r.Dir);
        }
    }
}
```

Exercícios

1. Implemente na classe ArvoreDeBusca um método em C# que retorne uma nova árvore, criada e formada pela cópia dos valores presentes na árvore original (this). A árvore retornada deve ter seus elementos dispostos de maneira a formar uma imagem espelhada da primeira.
2. Suponha que existe uma árvore binária que representa uma expressão aritmética. Em seus nós, existe um campo Info que armazena um operador (+, -, *, /, @) ou um operando (letra). Existe um vetor global que é indexado pelas letras do alfabeto e armazena o valor real correspondente a cada operando. O operador @ indica a operação de "menos unário". Um nó com esse operador terá apenas o filho direito, que representa a subexpressão que está sendo transformada em seu oposto.

Faça uma função que receba o apontador para a raiz dessa árvore e devolva o valor final da expressão.

Prova do 4º Bimestre de 1997:

3. Suponha que deve-se montar uma árvore de busca com os seguintes campos:

```
Conta : string[12]
Operação : ListaSimples
descendente_esquerdo : NoArvore
descendente_direito : NoArvore
numero_do_registro_no_arquivo : longint
```

Existe um arquivo de registros, já **aberto** e **ordenado** pelo campo Conta, que contém os dados da conta, a saber:

```
Conta : string[12]
Nome_do_Correntista: string[30]
Saldo_Inicial : real
```

Esse arquivo é muito grande e não cabe na memória. Portanto, usa-se a árvore para manter uma estrutura de indexação aos dados do arquivo. O campo numero_do_registro_no_arquivo de cada nó da árvore armazena o número do registro do arquivo de contas para acesso direto, quando necessário.

Um outro arquivo de registros, chamado arquivo de Operações, existe e está ordenado por Conta e Data. Ele deve ser lido sequencialmente e ter as **referências** aos registros de cada operação armazenados em uma lista ligada específica da conta na árvore, ou seja, usam-se os campos Primeira_Operacao e Ultima_Operacao para se incluir cada nó na lista ligada originária do nó da árvore. Assim, cada nó da árvore, além de permitir o acesso ao arquivo de contas, também listará as operações financeiras realizadas na conta. Cada nó da lista terá o código da conta, a data, o **número do registro no arquivo** e o próximo.

Portanto, após ter-se montado a estrutura, dever-se-á criar um procedimento que leia códigos de conta, procure-os na árvore e acesse seus dados no arquivo. A partir daí, deve-se emitir o extrato de operações dessa conta, de acordo com o lay-out abaixo:

```
-----  
Conta: xxxxxxxxxxxx      Correntista: yyyyyyyyyyyyyyyyyyyyyyyyy  
-----  
Data      Histórico          Tipo  Valor    Saldo  
          Saldo Anterior  
99/99/99  lams baesrtra dajiskad asti   D  999,99  9999999,99  
99/88/97  ehs lliw veah a ybab (eivom)  C   25,00  9999000,00  
           
                       
          Saldo Final          ???????, ??  
-----
```

O registro de operações contém os seguintes campos:

```
Conta      : string[12]
Data       : string [8]
Historico : string [30]
Tipo       : char      // D = débito; C = crédito
Valor     : real
```

1. (1996) Em um Centro de Processamento de Dados antigo, a permissão de uso de programas pelos usuários era feita da seguinte maneira: um usuário tinha que ser cadastrado, bem como os programas disponíveis no CPD. Em seguida, distribuía-se os programas que cada usuário poderia executar. Sempre que um usuário era excluído, os programas que ele podia executar também o eram. Sempre que um programa deixava de ser disponível para um usuário, isso tinha que ser registrado. Também caso houvesse um programa que deixasse de ser usado pelo CPD, todos os usuários com acesso a ele deveriam perder seu registro de permissão de uso. Para implementar isso, usava-se os arquivos:

Usuários	Programas	Permissões
ContaUsu String[8] NomeUsu String[30] Programas em Uso: Array[1..5] of String[6] Quantos Programas em Uso : Byte Manter em árvore binária os dados referentes à conta do usuário e a posição do registro no arquivo em disco.	CodProgr String[6] NomeProg String[9] Usuário com o programa : String[8] Deve-se usar uma árvore de busca para manter estes dados (não acessa diretamente o arquivo).	ContaUsu String[8] CodProgr String[6] Deve-se usar uma lista ligada ordenada pela concatenação dos dois campos para manter este arquivo

Efetuar:

Ao ler os arquivos, deve-se armazená-los nas estruturas de dados indicadas, criando árvores balanceadas quando for o caso. Para o nó da lista de Permissões, deve haver dois campos apontadores, um para o nó da árvore de Usuários, e outro para o nó da árvore de Programas, endereçando diretamente os registros associados à conta do usuário e ao código do programa. A ligação deve ser feita durante a leitura do arquivo de Permissões.

1 - Cadastrar Permissões - este arquivo indica, para os usuários que têm permissão de uso de programas, quais os códigos dos programas que ele pode executar. Deve-se ligar o nó de permissão aos nós correspondentes das demais árvores.

2 - Retirar Permissões - solicitar o programa; para cada usuário que tenha permissão para seu uso, retirar essa permissão, atualizando os apontadores envolvidos.

3 - Usar programas - deve-se solicitar o código do usuário que deseja usar um programa, e o código do programa. Em seguida, procurar na lista de permissões se é possível efetuá-la, e, a partir do nó desta lista, acessar os nós das duas árvores. Após isso, verificar se o usuário não tem mais de 5 programas já em seu poder, se o programa não está em poder de outro usuário e então registrar as referências em cada registro.

4 - Devolver programas - pede-se o código do usuário, e do programa que este quer devolver. Em seguida, procura-se os nós diretamente nas árvores e desfaz-se as referências nos registros do usuário e do programa.

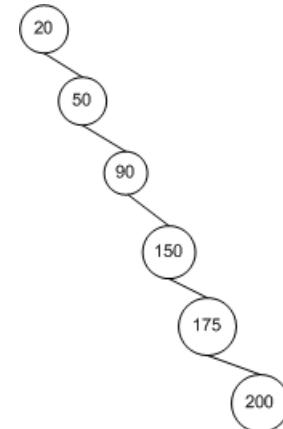
5 - Parar o programa, gravando os dados modificados nos arquivos correspondentes, na ordem das chaves de ordenação e acesso.

13. Balanceamento de Árvores

[[http://msdn.microsoft.com/en-us/library/ms379573\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms379573(VS.80).aspx)]

Árvores Binárias de Busca (ABB) — que oferecem tempo de execução sub-linear para inserções, remoções e pesquisas — funcionam otimamente quando seus nós são dispostos em uma maneira espalhada. Isto ocorre porque quando procurando por um nó nesse tipo de árvore, cada simples passo para baixo na árvore reduz pela metade o número de nós que potencialmente precisariam ser verificados.

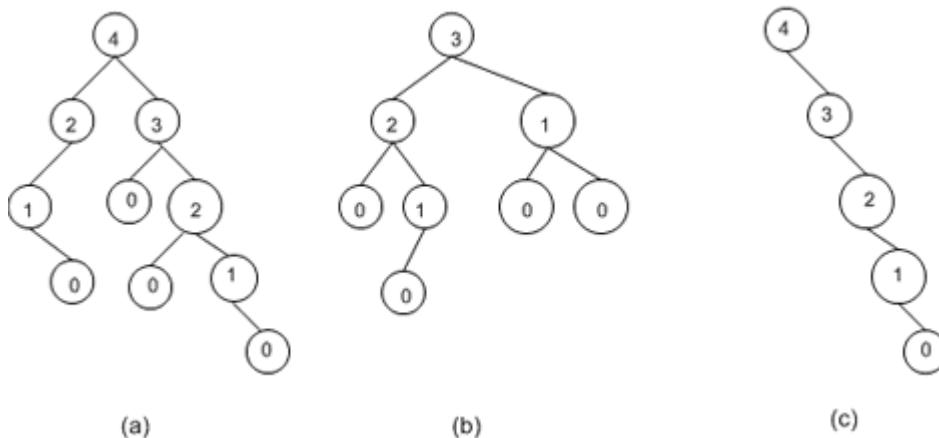
No entanto, quando uma ABB tem uma topologia similar à da figura ao lado, o tempo de execução para as operações em árvore são muito próximos do tempo linear $O(n)$ porque cada passo para o nível inferior da árvore apenas reduz em um o número de nós que precisariam ser pesquisados. Por exemplo, na figura, considere o que acontece quando pesquisamos o valor 175. Começando na raiz, 20, devemos navegar através de **cada filho direito** até que encontremos 175. Ou seja, não há economia em nós que precisam ser verificados a cada passo. Pesquisar uma árvore como a da figura 2 é idêntico à pesquisa sequencial em listas ligadas ou vetores— cada elemento deve ser verificado um a um, a cada vez. Portanto, uma ABB estruturada como essa exibirá um tempo linear de pesquisa $O(n)$, que é inferior ao potencial de $O(\log_2 n)$ de árvores de busca.



É importante perceber que o tempo de execução das operações em ABBs está relacionado com a *altura* da ABB. A altura de uma árvore é definida como o comprimento do caminho mais longo iniciando na raiz. A altura de uma árvore pode ser definida recursivamente como se segue:

- A altura de um nó sem filhos é 0 (folhas têm altura 0).
- A altura de um nó com um filho é a altura desse filho mais um.
- A altura de um nó com dois filhos é **um a mais que a maior entre as alturas** de cada um dos dois filhos.

Para calcular a altura de uma árvore, comece em seus nós folhas e atribua a eles uma altura de 0. Então mova-se para o nível anterior (para cima na árvore) usando as três regras acima aplicando-as a cada pai de um nó folha. Continue dessa maneira até que todos os nós da árvore tenham sido visitados. A altura da árvore, então, será a altura do nó raiz. A figura 3 ilustra um grupo de árvores binárias com suas alturas calculadas para cada nó. Os números em cada nó não são a chave do nó, e sim a sua altura. Observe que, nessa maneira de determinar a altura, todas as folhas possuem altura 0. A altura de cada um dos nós não-folhas é calculada, então, como sendo um mais a maior entre as alturas dos seus filhos.



Uma ABB possui tempo de execução $\log_2 n$ quando sua altura, definida a partir do número de nós n da árvore, é próximo ao maior inteiro menor que $\log_2 n$ (por exemplo, se $\log_2 n = 5.38$, o tempo de execução será 5). Das três árvores da figura acima, (b) tem a melhor relação entre número de nós e altura, pois há 8 nós e a altura da árvore é 3. Se rearranjássemos a árvore (a), ela poderia ter altura 3, caso tornássemos o nó mais à direita descendente de um dos nós com um único filho. Da forma como está, possui 10 nós e altura 4. No entanto, para uma árvore com 10 nós, a altura ótima em níveis é 3, pois $\log_2 10 = 3.3219$ e o maior inteiro menor que esse valor é 3. Finalmente, a árvore (c) tem a pior razão entre número de nós e altura. Com 5 nós, essa árvore poderia ter uma altura ótima de 2, mas devido à sua topologia linear, tem altura 4, o que torna a pesquisa duas vezes mais lenta do que seria necessário.

O desafio que temos, portanto, é assegurar que a topologia (estrutura) da árvore ABB resultante tenha uma razão ótima entre a altura e o número de nós.

Por ser a topologia de uma ABB baseada na ordem pela qual as chaves dos nós são inseridas, intuitivamente você poderia optar por resolver esse problema garantindo que os dados sejam adicionados em ordem aleatória. Poderá até mesmo ser possível que em determinadas situações você conheça os dados que serão adicionados à ABB, mas isso nem sempre será verdade e poderá nem mesmo ser prático. E, se você não sabe em que ordem os dados serão incluídos — como acontece, por exemplo, se eles são baseados em digitação pelo usuário, ou se são provenientes de leitura de um sensor — então não há como garantir que os dados não sejam inseridos numa ordem próxima à crescente ou decrescente. Em outras palavras, não há como garantir, a priori, que a árvore resultante das inclusões de chave não será degenerada.

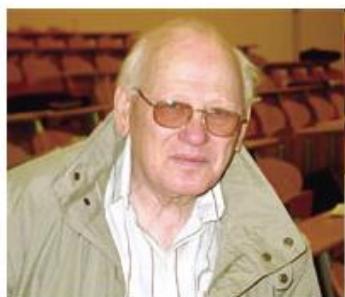
A solução, então, não é tentar ditar a ordem pela qual os dados serão inseridos, mas sim garantir que **após cada inserção a ABB permaneça balanceada**. Estruturas de Dados projetadas para manter o balanceamento são chamadas de árvores binárias de busca **auto-balanceadas**.

Uma árvore *balanceada* é uma árvore que mantém uma razão ótima entre sua altura e largura. Diferentes estruturas de dados definem suas próprias razões (proporções) para balanceamento, mas todas são próximas de $\log_2 n$. Uma ABB auto-balanceada, portanto, possui tempo de execução $O(\log_2 n)$. Há vários tipos de ABB auto-balanceadas, como árvores AVL, árvores vermelho-pretas, árvores 2-3, árvores 2-3-4, árvores splay, B-trees e outras.

Árvores AVL

Em 1962 os matemáticos russos G. M. Adel'son-Vel'skii e E. M. Landis inventaram a primeira ABB auto-balanceada, chamada de árvore AVL.

Adelson-Velskii & Landis



1922



1921

Essas árvores devem manter a seguinte propriedade de平衡amento: para cada nó X, a altura das subárvores esquerda e direita de X só pode diferir, no máximo, em 1. A altura da subárvore esquerda ou direita de um nó é a altura calculada para seu nó esquerdo ou direito usando a

técnica discutida na seção anterior. Se um nó possui apenas um filho, então a altura da subárvore sem filhos é definida como -1.

Através da **comparação contínua** entre as alturas das subárvores esquerda e direita de uma árvore, a árvore AVL sempre se apresenta balanceada. Elas usam uma técnica chamada **Rotação** para manter-se平衡adas.

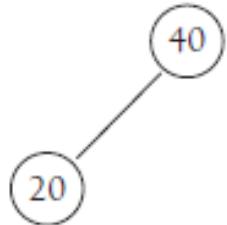


Figura 9.5

Para compreendermos como uma rotação funciona, vamos observar um exemplo simples que constrói uma árvore binária de inteiros. Começando com a árvore exibida na figura 9.5, se inserirmos o valor 10 na árvore, a árvore se torna desbalanceada, como vemos na figura 9.6. A subárvore esquerda agora tem uma altura de 2, mas a subárvore direita tem altura de 0, violando a regra para árvores AVL.

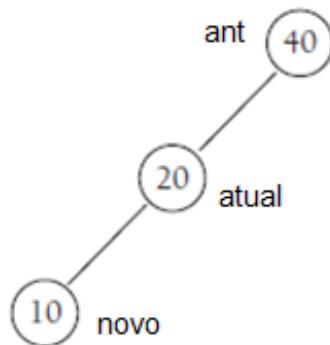


Figura 9.6

A árvore seria balanceada pela realização de uma **rotação simples à direita** (*single right rotation*, SRR), movendo o valor 40 para baixo à direita como vemos na figura 9.7.

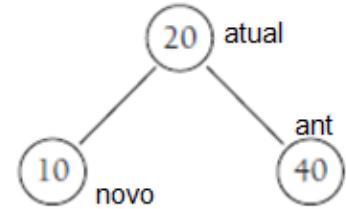
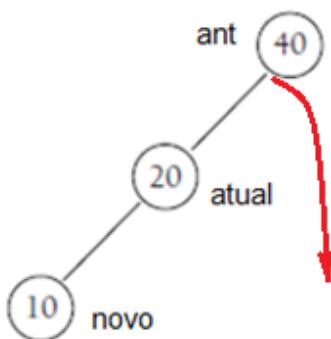


Figura 9.7

Agora observe a árvore na figura 9.8. Se inserirmos o valor 30, obteremos a árvore da figura 9.9. Essa árvore ficou desbalanceada após a inclusão.

Arrumamos tal situação realizando o que é chamado uma **rotação dupla**, movendo 40 para baixo à direita e 30 para cima, à direita, resultando na árvore que vemos na figura 9.10.

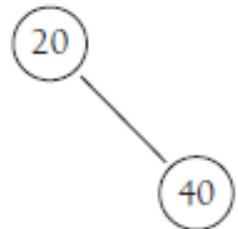


Figura 9.8

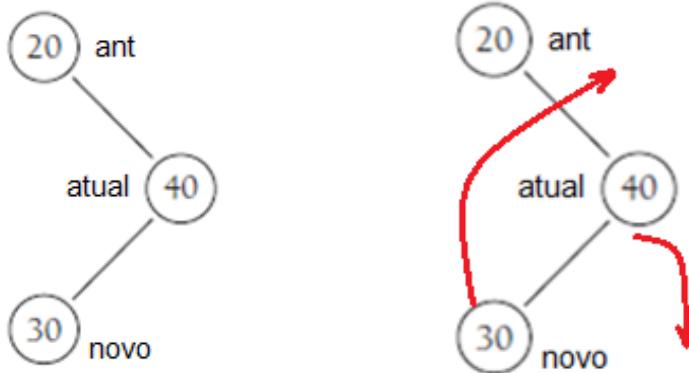


Figura 9.9

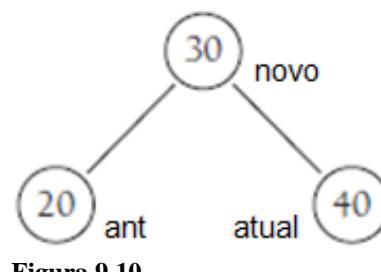
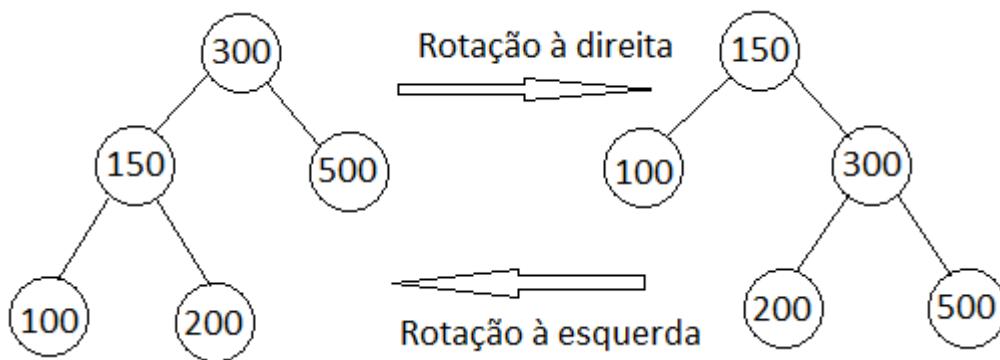


Figura 9.10



Passos para Inserção

Suponhamos que o **novo nó** a ser inserido é apontado por *w*.

1) Realize a **inserção padrão** de árvore binária de busca para o nó *w*.

2) Iniciando em *w*, percorra para cima e encontre o primeiro nó não balanceado, que chamaremos de *noAtual*; *temp* será o filho de *noAtual* que veio no caminho de *w* para *noAtual* e *x* será o neto de *noAtual* que vem no caminho de *w* para *noAtual*.

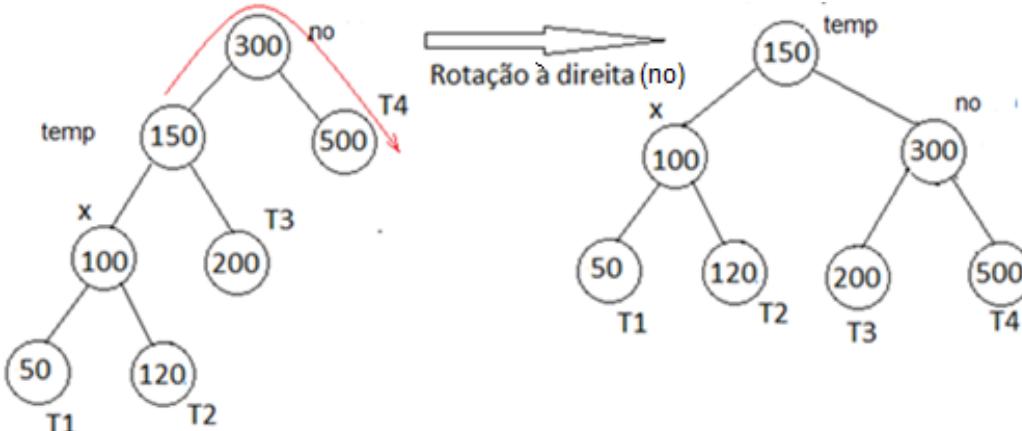
3) Rebalanceie a árvore realizando as rotações apropriadas na subárvore cuja raiz é *noAtual*. Há 4 casos possíveis que precisam ser tratados já que *x*, *temp* e *noAtual* podem ser dispostos de 4 maneiras. Estas são as disposições possíveis:

- a) *temp* é o filho esquerdo de *noAtual* e *x* é o filho esquerdo de *temp* (Caso Esquerdo Esquerdo)
- b) *temp* é o filho direito de *noAtual* e *x* é o filho direito de *temp* (Caso Direito Direito)
- c) *temp* é o filho esquerdo de *noAtual* e *x* é o filho direito de *temp* (Caso Esquerdo Direito)
- d) *temp* é o filho direito de *noAtual* and *x* é o filho esquerdo de *temp* (Caso Direito Esquerdo)

A seguir temos as operações que devem ser realizadas em cada um dos quatro casos mencionados acima. Em todos os casos, precisamos apenas rebalancear a subárvore enraizada em *noAtual* e a árvore completa volta a ser balanceada já que a altura da subárvore (após as rotações apropriadas) com raiz em *noAtual* fica a mesma que era antes da inserção.

(Veja a videoaula a <https://www.youtube.com/watch?v=TbhGcf6UJU> para provar).

a) Caso Esquerdo Esquerdo

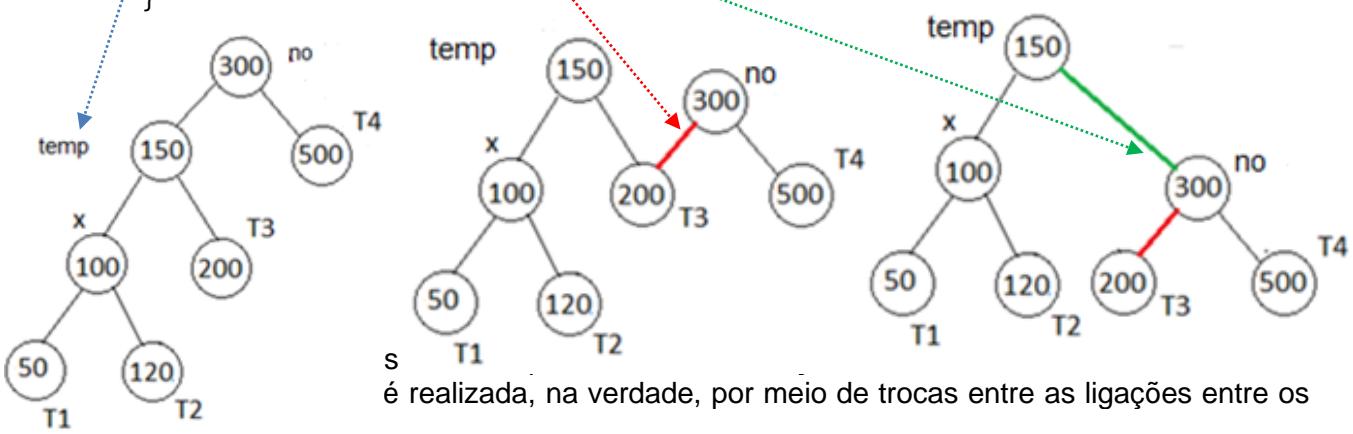


Acima, incluímos 50 ou 120 (*w*). **no** é o primeiro nó não balanceado acima do novo nó (*w*) e isso foi descoberto por meio do cálculo de alturas a partir do nó inserido e foi detectada uma diferença maior que 1 na altura das subárvores de **no**; **temp** é o **filho esquerdo** de **no** e **x** é o **neto**

esquerdo de **no** que vêm no caminho de **w** para **no**. Reorganizamos de forma que os descendentes da direita de **temp** (**T3**, pode haver mais de um descendente ai) passam a ser filhos esquerdos de **no**; **no** passa a ser filho direito de **temp** e **temp** é retornado como a **raiz** dessa subárvore.

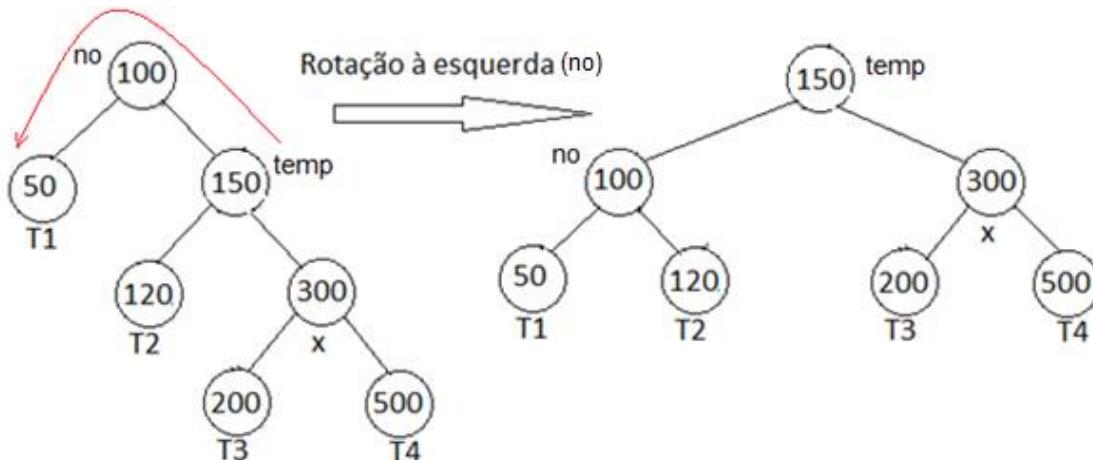
No novo método **InserirBalanceado()**, para árvores balanceadas, teremos esse código:

```
public NoArvore<Dado> InserirBalanceado(Dado item, NoArvore<Dado> noAtual)
{
    ...
    noAtual.Esq = InserirBalanceado(item, noAtual.Esq); // chamada recursiva
    if (getAltura(noAtual.Esq) - getAltura(noAtual.Dir) == 2) // getAltura() testa nulo!
        if (item.CompareTo(noAtual.Esq.Info) < 0)
            noAtual = RotacaoSimplexComFilhoEsquerdo (noAtual);
    ...
    private NoArvore<Dado> RotacaoSimplexComFilhoEsquerdo(NoArvore<Tipo> no)
    {
        NoArvore<Tipo> temp = no.Esq;
        no.Esq = temp.Dir; // temp.Dir é o nó apontado por x, na figura acima
        temp.Dir = no;
        no.altura = Math.Max(getAltura(no.esq), getAltura(no.dir)) + 1; // recalcula
        temp.altura = Math.Max(getAltura(temp.esq), getAltura(no)) + 1; // recalcula
        return temp;
    }
}
```



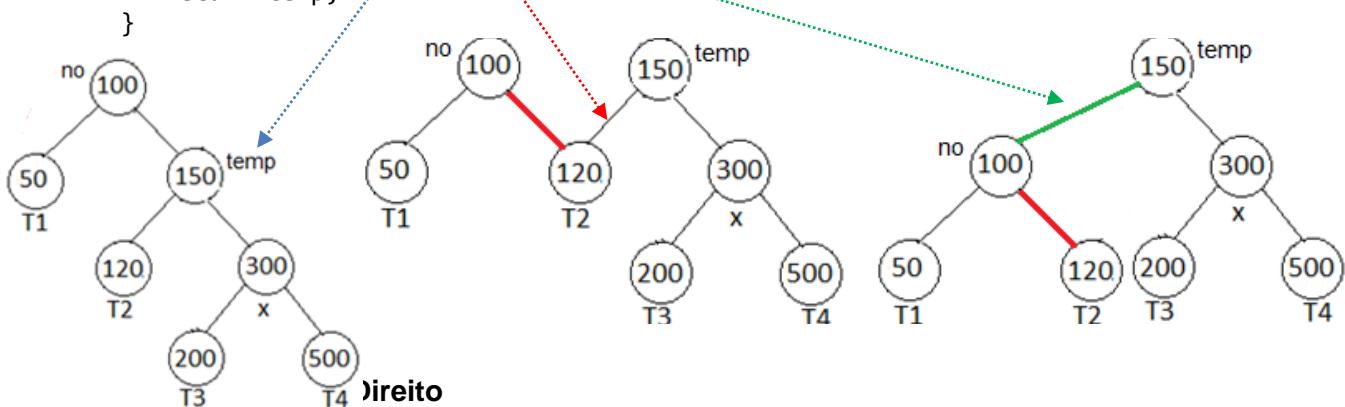
Caso haja um antecessor a **noAtual** (um nível acima), ele receberá o resultado da chamada recursiva, que será o ponteiro **temp**, que é retornado pelo método **RotacaoSimplexComFilhoEsquerdo()**. A variável que aponta **noAtual** num possível nível acima receberá um ponteiro para **temp** e, dessa forma, a árvore continuará com ligações que mantêm sua característica de árvore de busca.

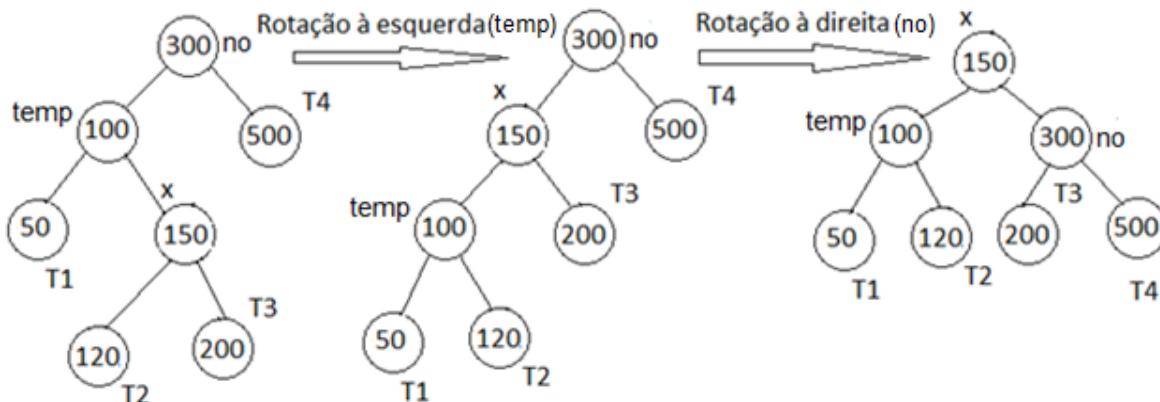
b) Caso Direito Direito



Acima, incluímos 200 ou 500 (w). **no** é o primeiro nó não balanceado acima do novo nó (w) e isso foi descoberto por meio do cálculo de alturas a partir do nó inserido e foi detectada uma diferença maior que 1 na altura das subárvore de **no**; **temp** é o filho direito de **no** e **x** é o neto direito de **no** que vêm no caminho de **w** para **no**. Reorganizamos de forma que os descendentes da esquerda de **temp** (T2, pode haver mais de um descendente ai) passam a ser filhos direitos de **no**; **no** passa a ser filho esquerdo de **temp** e **temp** é retornado como a **raiz** dessa sub-árvore. No novo método InserirBalanceado(), para árvores平衡adas, teremos esse código:

```
public NoArvore<Dado> InserirBalanceado(Dado item, NoArvore<Dado> noAtual)
{
    ...
    noAtual.Dir = InserirBalanceado(item, noAtual.Dir);
    if (getAltura(noAtual.Dir) - getAltura(noAtual.Esq) == 2) // getAltura testa nulo!
        if (item.CompareTo(noAtual.Dir.Info) > 0)
            noAtual = RotacaoSimplexComFilhoDireito(noAtual);
    ...
    private NoArvore<Dado> RotacaoSimplexComFilhoDireito(NoArvore<Dado> no)
    {
        NoArvore<Dado> temp = no.Dir;
        no.Dir = temp.Esq;
        temp.Esq = no;
        no.Altura = Math.Max(getAltura(no.Esq), getAltura(no.Dir)) + 1; // recalcula
        temp.Altura = Math.Max(getAltura(temp.Dir), getAltura(no)) + 1; // recalcula
        return temp;
    }
}
```



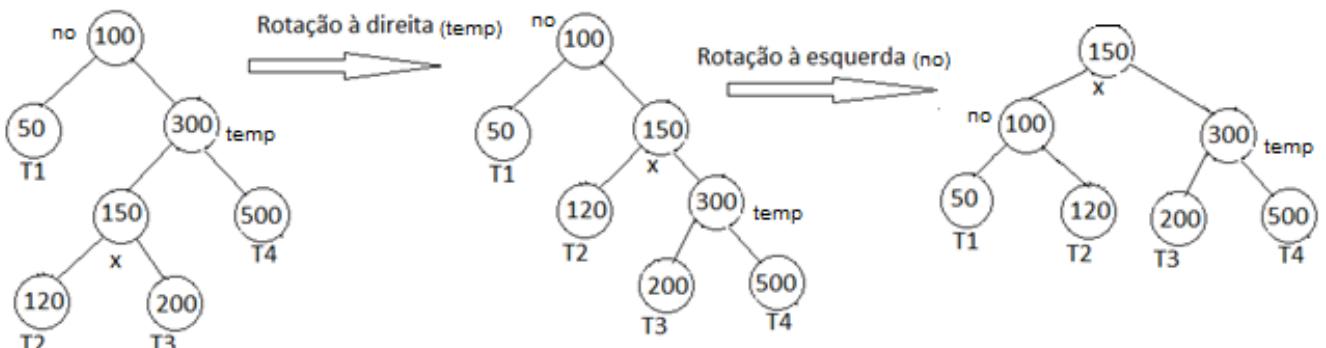


Acima, incluímos 120 ou 200 (w). **no** é o primeiro nó não balanceado acima do novo nó (w); **temp** e **x** são, respectivamente, o **filho** e o **neto** de **no** que vêm no caminho de w para **no**. Observen que ocorreu uma mudança de direção, da esquerda para a direita, no caminho entre **no** e w. Reorganizamos chamando o método de `RotacaoSimplesComFilhoDireito()` e, depois, `RotacaoSimplesComFilhoEsquerdo()`, que alterarão os ponteiros necessários. A essa sequência chamamos de `RotacaoDuplaComFilhoEsquerdo`, e o código seria o seguinte:

```
public NoArvore<Dado> InserirBalanceado(Dado item, NoArvore<Dado> noAtual)
{
    ...
    if (item.CompareTo(noAtual.Info) < 0)
    {
        noAtual.Esq = InserirBalanceado(item, noAtual.Esq);
        if (getAltura(noAtual.Esq) - getAltura(noAtual.Dir) == 2) // getAltura testa nulo!
            if (item.CompareTo(noAtual.Esq.Info) < 0)
                noAtual = RotacaoSimplesComFilhoEsquerdo(noAtual);
            else
                noAtual = RotacaoDuplaComFilhoEsquerdo(noAtual);
    }
    ...
    private NoArvore<Dado> RotacaoDuplaComFilhoEsquerdo(NoArvore<Dado> no)
    {
        no.Esq = RotacaoSimplesComFilhoDireito( no.Esq );
        return RotacaoSimplesComFilhoEsquerdo(no);
    }
}
```

Siga o fluxo dessas chamadas e compare com a figura acima para observar como as ligações são refeitas.

e) Caso Direito Esquerdo



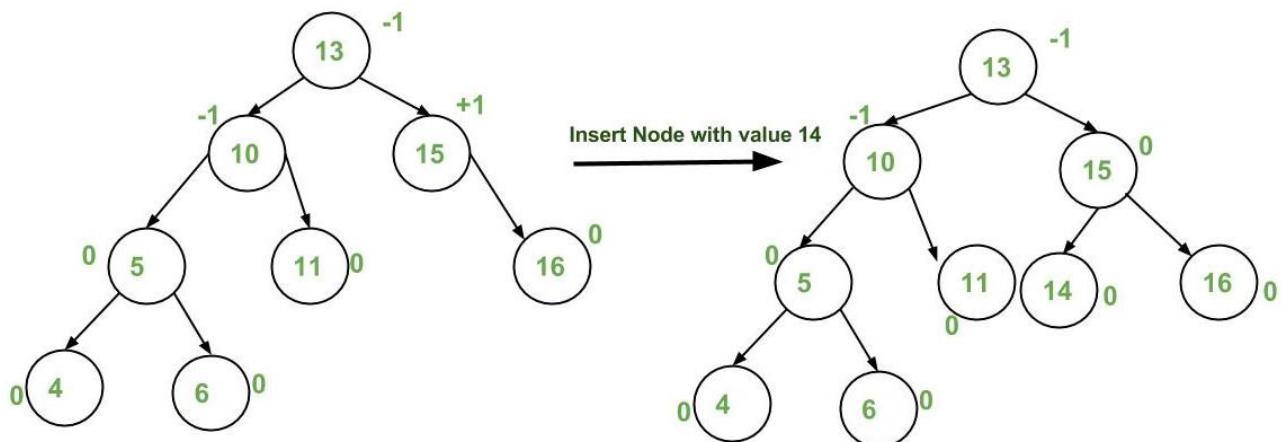
Acima, incluímos 120 ou 200 (w). **no** é o primeiro nó não balanceado acima do novo nó (w); **temp** e **x** são, respectivamente, o **filho** e o **neto** de **no** que vêm no caminho de w para **no**. Observen que ocorreu uma mudança de direção, da direita para a esquerda, no caminho entre **no** e w.

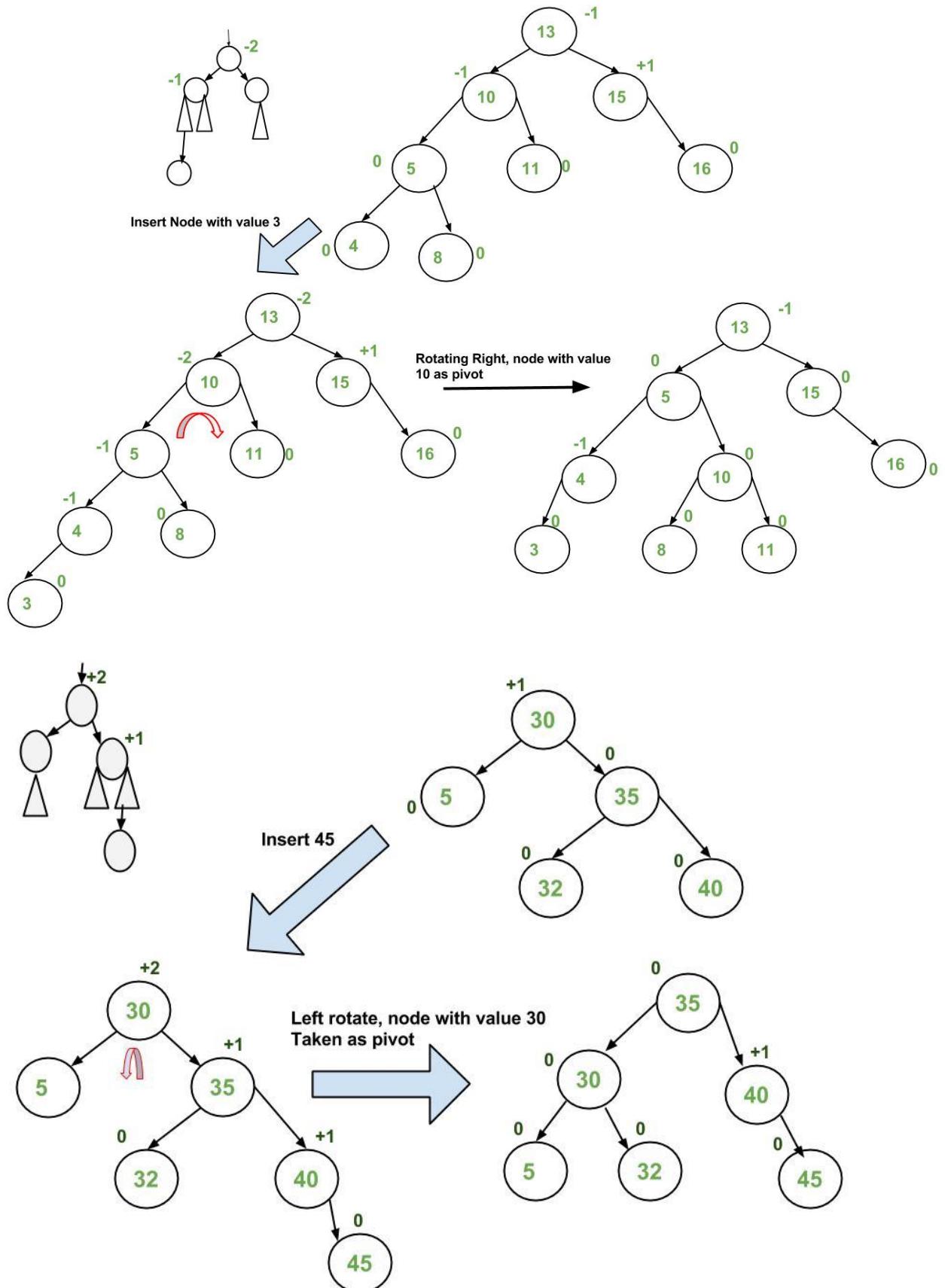
Reorganizamos chamando o método de `RotacaoSimplesComFilhoEsquerdo()` e, depois, `RotaçãoSimplesComFilhoDireito()`, que alterarão os ponteiros necessários. A essa sequência chamamos de `RotacaoDuplaComFilhoDireito`, e o código seria o seguinte:

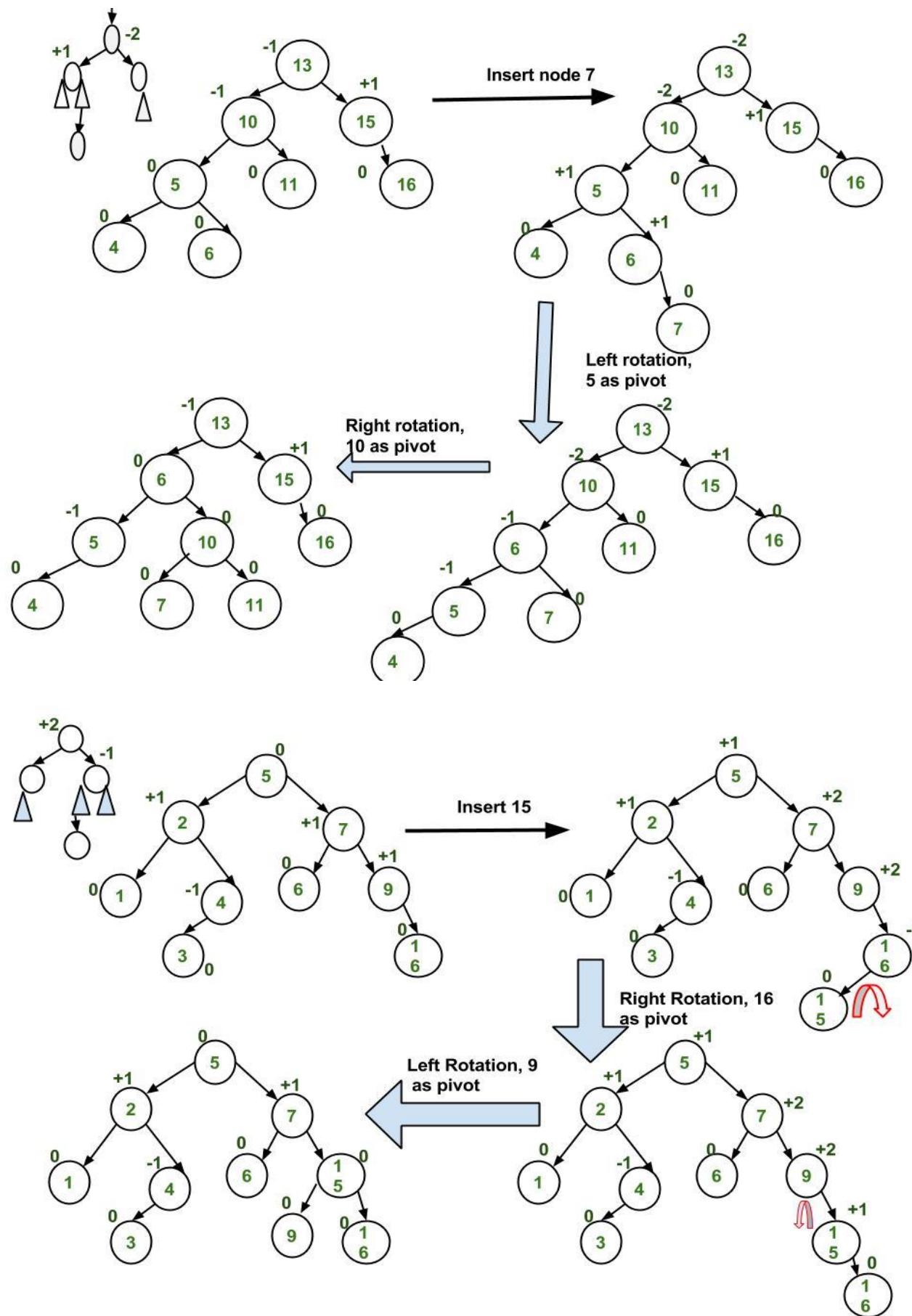
```
public NoArvore<Dados> InserirBalanceado(Dado item, NoArvore<Dados> noAtual)
{
    if (item.CompareTo(noAtual.Info) > 0)
    {
        noAtual.Dir = InserirBalanceado(item, noAtual.Dir);
        if (getAltura(noAtual.Dir) - getAltura(noAtual.Esq) == 2) // getAltura testa nulo!
            if (item.CompareTo(noAtual.Dir.Info) > 0)
                noAtual = RotacaoSimplesComFilhoDireito(noAtual);
            else
                noAtual = RotacaoDuplaComFilhoDireito(noAtual);
    }
    ...
    private NoArvore<Dados> RotacaoDuplaComFilhoDireito(NoArvore<Dados> no)
    {
        no.Dir = RotacaoSimplesComFilhoEsquerdo( no.Dir );
        return RotacaoSimplesComFilhoDireito( no );
    }
}
```

Siga o fluxo dessas chamadas e compare com a figura acima para observar como as ligações são refeitas.

Exemplos de inserção (do link acima)







A implementação da Árvore AVL

A implementação de nossa árvore AVL consiste de duas classes: uma nova classe NoArvore usada para manter dados para cada nó da árvore e uma nova classe ArvoreDeBusca, que contém os métodos para inserir e rotacionar nós.

A classe NoArvore para a implementação da árvore AVL é construída de forma similar aos nós da implementação de uma árvore binária, mas com uma diferença importante. Cada nó em uma árvore AVL deve **conter dados sobre sua altura**, de forma que um membro de dados para altura é incluído na classe. Essa classe também implementará a interface IComparable para que possamos comparar os valores (e suas chaves de pesquisa, se for o caso) armazenados nos nós.

Segue abaixo o código para a nova classe NoArvore:

```
namespace ArvoreBinaria
{
    class NoArvore<Dado> : IComparable<NoArvore< Dado>> where Dado : IComparable<Dado>
    {
        private Dado info;
        private NoArvore<Dado> esq;
        private NoArvore<Dado> dir;
        private int altura;
        private bool estaMarcadoParaMorrer;

        public Dado Info { get => info; set => info = value; }
        internal NoArvore<Dado> Esq { get => esq; set => esq = value; }
        internal NoArvore<Dado> Dir { get => dir; set => dir = value; }
        public int Altura { get => altura; set => altura = value; }
        public bool EstaMarcadoParaMorrer
            { get => estaMarcadoParaMorrer; set => estaMarcadoParaMorrer = value; }

        public NoArvore(Dado dados)
        {
            this.Info = dados;
            this.Esq = null;
            this.Dir = null;
            this.Altura = 0;
        }

        public NoArvore(Dado dados, NoArvore<Dado> esquierdo, NoArvore<Dado> direito,
                        int altura)
        {
            this.Info = dados;
            this.Esq = esquierdo;
            this.Dir = direito;
            this.Altura = altura;
        }

        public int CompareTo(NoArvore<Dado> o)
        {
            return Info.CompareTo(o.Info);
        }

        public bool Equals(NoArvore<Dado> o)
        {
            return this.Info.Equals(o.Info);
        }
    }
}
```

O primeiro método da nova classe ArvoreDeBusca que examinaremos é o método InserirBalanceado(). Este método determina onde colocar um nó na árvore. Ele é recursivo, tanto movendo para a esquerda quando o nó atual é maior que o nó que será inserido quanto movendo para a direita quando o nó atual é menor que o nó a ser inserido.

Uma vez que o novo nó esteja em seu lugar, a **diferença de altura das duas subárvore**s é calculada. Se é determinado que a árvore ficou desbalanceada, uma rotação para a esquerda ou para a direita, ou ainda uma rotação dupla para a esquerda ou dupla para a direita é realizada. Além disso, devido ao fato de a altura de um nó ser tão importante, incluiremos um método (ou uma propriedade) para retornar a altura de um nó.

Segue o código (o código para as diversas rotações é mostrado após o método InserirBalanceado()):

```
public int getAltura(NoArvore<Tipo> no)
{
    if (no != null)
        return no.altura;
    else
        return -1;
}

public NoArvore<Dado> InserirBalanceado(Dado item, NoArvore<Dado> noAtual)
{
    if (noAtual == null)
        noAtual = new NoArvore<Dado>(item);
    else
    {
        if (item.CompareTo(noAtual.Info) < 0)
        {
            noAtual.Esq = InserirBalanceado(item, noAtual.Esq);
            if (getAltura(noAtual.Esq)-getAltura(noAtual.Dir) == 2) // getAltura testa nulo
                if (item.CompareTo(noAtual.Esq.Info) < 0)
                    noAtual = RotacaoSimplesComFilhoEsquerdo(noAtual);
                else
                    noAtual = RotacaoDuplaComFilhoEsquerdo(noAtual);
        }
        else
            if (item.CompareTo(noAtual.Info) > 0)
            {
                noAtual.Dir = InserirBalanceado(item, noAtual.Dir);
                if (getAltura(noAtual.Dir)-getAltura(noAtual.Esq) == 2) // getAltura testa nulo
                    if (item.CompareTo(noAtual.Dir.Info) > 0)
                        noAtual = RotacaoSimplesComFilhoDireito(noAtual);
                    else
                        noAtual = RotacaoDuplaComFilhoDireito(noAtual);
            }
        //else ; - não faz nada, valor duplicado
        noAtual.Altura = Math.Max(getAltura(noAtual.Esq), getAltura(noAtual.Dir)) + 1;
        OndeExibir.Invalidate();
        Application.DoEvents();
    }
    return noAtual;
}
```

Os diversos métodos de rotação seguem abaixo:

```
private NoArvore<Tipo> RotacaoSimplesComFilhoEsquerdo(NoArvore<Tipo> no)
{
```

```
NoArvore<Tipo> temp = no.Esq;
no.Esq = temp.Dir;
temp.Dir = no;
no.Altura = Math.Max(getAltura(no.Esq), getAltura(no.Dir)) + 1;
temp.Altura = Math.Max(getAltura(temp.Esq), getAltura(no)) + 1;
return temp;
}

private NoArvore<Tipo> RotacaoSimplesComFilhoDireito(NoArvore<Tipo> no)
{
    NoArvore<Tipo> temp = no.Dir;
    no.Dir = temp.Esq;
    temp.Esq = no;
    no.Altura = Math.Max(getAltura(no.Esq), getAltura(no.Dir)) + 1;
    temp.Altura = Math.Max(getAltura(temp.Dir), getAltura(no)) + 1;
    return temp;
}

private NoArvore<Tipo> RotacaoDuplaComFilhoEsquerdo(NoArvore<Tipo> no)
{
    no.Esq = RotacaoSimplesComFilhoDireito( no.Esq );
    return RotacaoSimplesComFilhoEsquerdo(no);
}

private NoArvore<Tipo> RotacaoDuplaComFilhoDireito(NoArvore<Tipo> no)
{
    no.Dir = RotateWithLeftChild( no.Dir );
    return RotacaoSimplesComFilhoDireito( no );
}
```

Há muitos outros métodos que podemos implementar para esta classe, ou seja, os métodos da classe ArvoreDeBusca. Deixamos a implementação desses métodos como projeto. Além disso, de propósito não implementamos um método de remoção de nós da nova classe ArvoreDeBusca com comportamento de árvore AVL. Em nossa aplicação de exemplo, há vários comandos nesses métodos para podermos ver o que está acontecendo, através de Messageboxes.

<https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

Pesquisa: Exclusão de nós em árvore AVL

<https://www.geeksforgeeks.org/avl-tree-set-2-deletion/>

Várias implementações de árvores AVL usam *lazy deletion*. Este sistema de remoção de nós marcam um nó para remoção futura mas não o remove realmente da árvore, pois o custo em desempenho da remoção de nós e posterior rebalanceamento da árvore é proibitivo.

Escreva um método Delete para a classe ArvoreAVL que use lazy deletion. Há muitas técnicas que você pode usar, mas a mais simples é adicionar um atributo boolean na classe NoArvoreAVL que informe se o nó está ou não marcado para remoção. Os outros métodos da classe devem ser modificados para levar em conta este campo e seu significado no funcionamento da árvore.

http://www.ondotnet.com/pub/a/dotnet/excerpt/progsharp4_ch09-04/index.html?page=2



<http://www.codeproject.com/KB/cs/objserial.aspx>

14. Como serializar classes criadas pelo programador em C#

Introdução

Gravar dados críticos em disco como texto sempre é perigoso. Qualquer usuário desconhecido pode abrir o arquivo texto e facilmente ler seus dados. Com a Serialização de Objetos, você pode reduzir este risco. Você pode gravar qualquer objeto complexo diretamente num filestream (sequência de bytes) sem converter para texto os valores das propriedades individuais do objeto. Assim, você pode tornar os dados escritos no disco ao menos não legíveis diretamente por um ser humano. A fim de que usuários possam ler seus arquivos de dados, eles precisarão usar o seu programa que gerou o arquivo em formato serializado.

.NET e Serialização de Objetos

Serialização é o processo de converter objetos complexos em sequências de bytes para armazenamento (ou transferência por meios de comunicação, como redes). Dessa serialização é o processo inverso, que corresponde a desempacotar a sequência de bytes na sua forma original. O namespace usado para ler e gravar arquivos é System.IO. Para Serialização nós usaremos o namespace System.Runtime.Serialization. A interface ISerializable permite que façamos com que qualquer classe se torne serializável.

Eis aqui os passos que tomaremos para criar uma classe serializável e testá-la.

- Crie uma classe chamada Funcionario e atribua propriedades a ela;
- Defina as funções de serialização;
- Crie uma classe main e instancie objetos da classe Funcionario;
- Serialize os objetos em um arquivo de amostra;
- Dessa serializar os valores lendo-os desse arquivo;

Definindo a classe Funcionario e suas propriedades

A nossa classe Funcionario deve ser derivada da interface ISerializable e deve ter o atributo [Serializable()] , como vemos no código abaixo:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace WindowsFormsApplication1
{
    [Serializable()] // Configure este atributo em todas as classes que queira serializar
    class Funcionario : ISerializable // derive sua classe a partir de ISerializable
    {
        public int matricula;
        public string nome;

        public Funcionario()
        {
            matricula = 0;
            nome = null;
        }
    }
}
```

Defina as funções de Serialização

Agora precisamos de duas funções: uma para especificar como serializar objetos da classe Funcionario e outra para especificar como os desserializar. Para a serialização nós substituímos a função GetObjectData() fornecida pela interface ISerializable. Para a desserialização, forneceremos um construtor especial com os parâmetros de serialização como argumentos. Este construtor será chamado quando nós desserializamos nosso arquivo para objetos (isso será mostrado mais à frente).

Um dos parâmetros importantes é o objeto SerializationInfo. Este objeto armazena um par nome-valor para as propriedades que serão serializadas. Você pode decidir quais propriedades deverão ser serializadas e quais não o serão, na função GetObjectData(). Todas as propriedades adicionadas a este parâmetro SerializationInfo serão serializadas. Abaixo temos os códigos para as duas funções. Adicione-os à sua classe Funcionario:

```
//Construtor de Desserialização
public Funcionario(SerializationInfo info, StreamingContext ctxt)
{
    //Get the values from info and assign them to the appropriate properties
    matricula = (int)info.GetValue("numeroMatricula", typeof(int));
    nome = (String)info.GetValue("nomeFuncionario", typeof(string));
}

//Função de Serialização
public void GetObjectData(SerializationInfo info, StreamingContext ctxt)
{
    // você pode usar qualquer nome que escolha para os pares nome-valor. Mas garanta que
    // sejam recuperados com o mesmo nome. Por ex:- se você gravou matricula com o nome
    // "numeroMatricula" então o valor deverá ser recuperado com o mesmo nome
    // "numeroMatricula"
    info.AddValue("numeroMatricula", matricula);
    info.AddValue("nomeFuncionario", nome);
}
```

É isso. Você criou sua própria classe serializável. Agora vejamos como gravar uma instância de Funcionario em um arquivo especial com uma extensão (tipo) .osl. E veremos também como recuperar um Funcionario armazenado nesse arquivo.

Instancie um objeto da classe Funcionario. Isso pode ser feito em um formulário de um projeto Windows Forms ou no método main de uma aplicação Console, como vemos abaixo:



```
Funcionario umFunc = null;

// Abre um arquivo e serializa o objeto em formato binário.
// FuncInfo.osl é o arquivo que estamos criando ou abrindo.
// Nota:- você pode dar qualquer extensao que queira ao nome do arquivo
// Se você usar extensões não-padrão, o usuário saberá que o arquivo
// está associado ao seu programa.
```

```
Stream stream = File.Open("FuncInfo.osl", FileMode.OpenOrCreate);
BinaryFormatter bformatter = new BinaryFormatter();

bool primeiraLeitura = true;

private void btnSerializar_Click(object sender, EventArgs e)
{
    umFunc = new Funcionario();
    umFunc.matricula = Convert.ToInt32(txtMatricula.Text);
    umFunc.nome = txtNome.Text;
}
```

Serialize o objeto em um arquivo de amostra

Para serializar, vamos abrir um objeto stream e dar-lhe o nome de um arquivo de amostra FuncInfo.osl. Quando você executar seu programa, o arquivo FuncInfo.osl será criado sob a mesma pasta onde você copiar o arquivo executável do programa. Adicione o código abaixo ao evento onclick do botão btnSerializar. Uma vez que o stream esteja aberto, nós criamos um objeto da classe BinaryFormatter e usamos o seu método Serialize para serializar nosso objeto no stream. O que o método Serialize fará? Ele converterá nosso objeto em formato binário e o escreve sequencialmente no stream, byte a byte.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

...
umFunc.nome = txtNome.Text;

this.Text = "Gravando informações de Funcionário";
bformatter.Serialize(stream, umFunc);
}
```

Desserializando os valores ao lê-los do arquivo

Agora leremos o arquivo criado e converteremos para a nossa classe Funcionario os valores retornados, para usá-los. Para a leitura criaremos novamente um BinaryFormatter para ler o objeto em formato binário. Em seguida usaremos o método Deserialize, que converte a sequência de bytes para um objeto da classe Object. Este objeto, então, poderá ser facilmente convertido para a classe Funcionario por meio de Cast.

```
private void btnLer_Click(object sender, EventArgs e)
{
    // Limpa umFunc para uso posterior
    umFunc = null;

    // Abre o arquivo gravado e lê valores a partir dele
    if (primeiraLeitura)
    {
        stream.Close();
        stream = File.Open("FuncInfo.osl", FileMode.Open);
    }

    primeiraLeitura = false;

    this.Text = "Lendo Informações de Funcionário";
    try
    {
        umFunc = (Funcionario)bformatter.Deserialize(stream);
```

```
        txtMatricula.Text = umFunc.matricula.ToString();
        txtNome.Text = umFunc.nome;
    }
    catch
    {
        this.Text = "Fim da leitura";
        primeiraLeitura = true;
    }
}

// O evento abaixo fecha o arquivo stream quando deixamos o formulário
private void frmSerializacao_Leave(object sender, EventArgs e)
{
    stream.Close();
}
```

Conclusão

Esta aplicação de amostra explica apenas a parte principal da serialização. Na realidade, você pode fazer muito mais coisas com os objetos enquanto os serializa e desserializa.

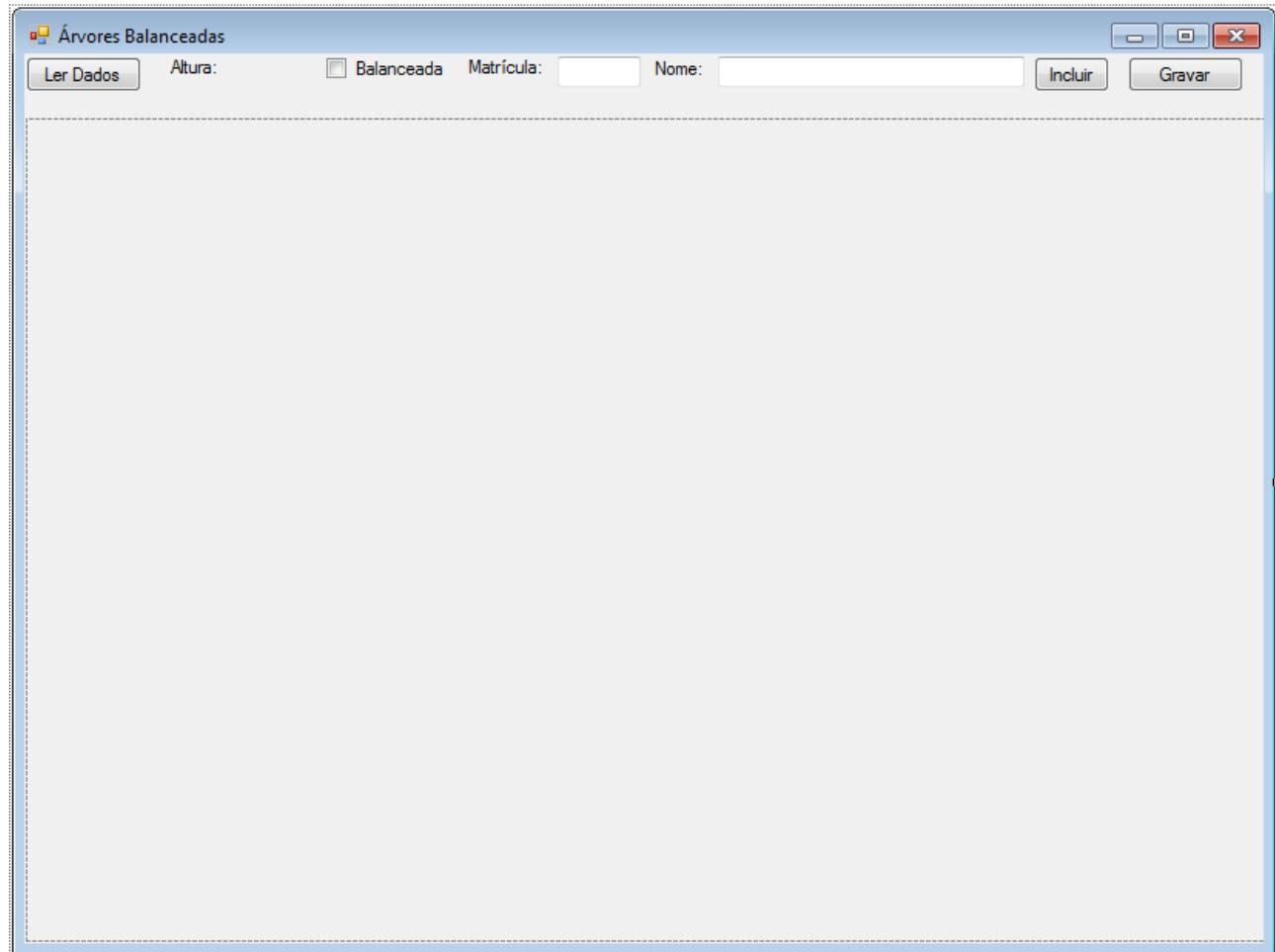
About the Author

omkamal Started programming with computers in 1995. Ever since it keeps me busy and Web Developer creative. Did a bachelor's degree in electronics and communication only to better understand the inside of computers and its power. Currently working as a software  United States developer in US and looking for a girl-friend...

Este método de armazenamento de objetos em arquivos pode ser usado para gravar em disco os elementos armazenados em nós de uma lista ligada ou de uma árvore binária. No entanto, o arquivo gerado possui "registros" de tamanho variável, pois as strings armazenadas podem ter até 2Gb de comprimento, sem tamanho fixo. Dessa maneira, não é possível tratar arquivos de objetos serializados como arquivos de acesso aleatório, onde indicamos o número do registro desejado e o lemos diretamente.

Isso significa que, para podermos tratar um arquivo como arquivo de acesso aleatório, temos que garantir que os dados que gravamos sempre utilizam um número fixo de bytes. Assim, sabendo o tamanho fixo que todos os registros possuem nesse arquivo, podemos facilmente calcular o número do byte inicial de cada registro e usar o método FileStream.Seek para posicionar a leitura nesse byte. O objeto que leremos é que deverá ler cada campo individualmente, pois a aplicação não tem que saber os detalhes de implementação do registro.

Lendo dados serializados e armazenando-os em árvores AVL



Vamos criar um projeto Windows Form e colocar os componentes visualizados na figura acima, que são um botão btnLerDados, um Label lblAltura, um checkBox chkBalanceada, dois Labels com Text iguais a Matricula e Nome, respectivamente, dois TextBoxes nomeados txtMatricula e txtNome e mais dois botões, nomeados btnIncluir e btnGravar. Vamos também colocar um panel sob esses componentes, chamado pnlArvore.

Vamos copiar o arquivo ArvoreDeBusca.cs na pasta do projeto, adicioná-lo ao projeto através do Solution Explorer e fazer as alterações abaixo:

```
namespace ArvoreBinaria
{
    class ArvoreDeBusca<Tipo> : IComparable<NoArvore<Tipo>>
        where Tipo : IComparable<Tipo>
    {
        public NoArvore<Tipo> raiz,
            atual,
            antecessor;

        public Panel painelArvore;

        public Panel OndeExibir
        {
            get { return painelArvore; }
            set { painelArvore = value; }
        }
    }
}
```

Vamos também copiar o arquivo Funcionario.cs que usamos na discussão sobre Serialização e adicioná-lo ao projeto. Abaixo temos o código desse arquivo, já com as alterações necessárias para que seus objetos possam ser armazenado em nós da árvore AVL, alterações essas marcadas nos quadros abaixo (`IComparable<Funcionario>` e método `compareTo`):

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace ArvoreBinaria
{
    [Serializable()] // Configure este atributo em todas as classes que queira serializar
    class Funcionario : ISerializable, IComparable<Funcionario> //derive sua classe a partir
    de ISerializable
    {
        public int matricula;
        public string nome;

        public Funcionario()
        {
            matricula = 0;
            nome = null;
        }

        public int CompareTo(Funcionario outroFunc)
        {
            return matricula - outroFunc.matricula;
        }

        //Construtor de Dessaerialização
        public Funcionario(SerializationInfo info, StreamingContext ctxt)
        {
            //Get the values from info and assign them to the appropriate properties
            matricula = (int)info.GetValue("numeroMatricula", typeof(int));
            nome = (String)info.GetValue("nomeFuncionario", typeof(string));
        }

        //Função de Serialização
        public void GetObjectData(SerializationInfo info, StreamingContext ctxt)
        {
        // você pode usar qualquer nome que escolha para os pares nome-valor. Mas garanta que sejam
        // recuperados com o mesmo nome. Se você gravou matricula com o nome "numeroMatricula"
        // então o valor deverá ser recuperado com o mesmo nome "numeroMatricula"
            info.AddValue("numeroMatricula", matricula);
            info.AddValue("nomeFuncionario", nome);
        }
    }
}
```

No código fonte do formulário, codifique as declarações abaixo:

```
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

namespace ArvoreBinaria
{
    public partial class Form1 : Form
    {

        ArvoreDeBusca<Funcionario> arvore = new ArvoreDeBusca<Funcionario>();
```

```
// Abre um arquivo e serializa o objeto em formato binário.  
// FuncInfo.osl é o arquivo que estamos criando ou abrindo.  
// Nota:- você pode dar qualquer extensão que queira ao nome do arquivo  
// Se você usar extensões não-padrão, o usuário saberá que o arquivo  
// está associado ao seu programa.  
  
Stream stream;  
BinaryFormatter bformatter = new BinaryFormatter();  
  
public Form1()  
{  
    InitializeComponent();  
    arvore.OndeExibir = pnlArvore;  
}
```

Digite o método DesenhaArvore como abaixo:

```
private void desenhaArvore(bool primeiraVez, NoArvore<Funcionario> raiz,  
                           int x, int y, double angulo, double incremento,  
                           double comprimento, Graphics g)  
{  
    int xf, yf;  
    if (raiz != null)  
    {  
        Pen caneta = new Pen(Color.Red);  
        xf = (int) Math.Round(x + Math.Cos(angulo) * comprimento);  
        yf = (int) Math.Round(y + Math.Sin(angulo) * comprimento);  
        if (primeiraVez)  
            yf = 25;  
        g.DrawLine(caneta, x, y, xf, yf);  
        // sleep(100);  
        desenhaArvore(false, raiz.esq, xf, yf, Math.PI/2 + incremento,  
                      incremento*0.60, comprimento*0.8, g );  
        desenhaArvore(false, raiz.dir, xf, yf, Math.PI/2 - incremento,  
                      incremento*0.60, comprimento*0.8, g );  
        // sleep(100);  
        SolidBrush preenchimento = new SolidBrush(Color.Blue);  
        g.FillEllipse(preenchimento, xf-15, yf-15, 30, 30);  
        g.DrawString(Convert.ToString(raiz.dados.matricula), new Font("Comic Sans",12),  
                    new SolidBrush(Color.Yellow), xf-15,yf-10);  
    }  
}
```

O Panel terá seu evento Paint tratado pelo código abaixo:

```
private void pnlArvore_Paint(object sender, PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    desenhaArvore(true, arvore.Raiz, (int)pnlArvore.Width / 2, 0, Math.PI / 2,  
                  Math.PI / 2.5, 300, g);  
}
```

O botão btnIncluir terá o evento onClick tratado pelo código abaixo:

```
private void btnIncluir_Click(object sender, EventArgs e)  
{  
    Funcionario umFunc = new Funcionario();  
    umFunc.matricula = Convert.ToInt32(txtChave.Text);  
    umFunc.nome = txtNome.Text;  
    arvore.raiz = arvore.Insert(umFunc, arvore.raiz);  
    pnlArvore.Invalidate();  
}
```

}

O botão btnGravar terá o evento onClick tratado pelo código abaixo:

```
private void btnGravar_Click(object sender, EventArgs e)
{
    stream = File.Open("FuncInfo.osl", FileMode.Create);
    gravarInOrdem(arvore.Raiz);
    stream.Close();
}

private void gravarInOrdem(NoArvore<Funcionario> r)
{
    if (r != null)
    {
        gravarInOrdem(r.esq);

        this.Text = "Gravando informações do Funcionário " + r.dados.matricula;
        Funcionario umFunc = new Funcionario();
        umFunc.matricula = Convert.ToInt32(r.dados.matricula);
        umFunc.nome = r.dados.nome;
        bformatter.Serialize(stream, umFunc);

        gravarInOrdem(r.dir);
    }
}
```

O evento que trata o botão btnLerDados terá o código abaixo:

```
private void btnLerDados_Click(object sender, EventArgs e)
{
    // ler do stream serializado

    // Limpa umFunc para uso posterior
    Funcionario umFunc;

    // Abre o arquivo gravado e lê valores a partir dele
    stream = File.Open("FuncInfo.osl", FileMode.Open);
    bool podeLer = true;

    this.Text = "Lendo Informações de Funcionário";

    while (podeLer)
        try
    {
        umFunc = (Funcionario)bformatter.Deserialize(stream);
        arvore.raiz = arvore.Insert(umFunc, arvore.raiz);
        pnlArvore.Invalidate();
    }
    catch
    {
        this.Text = "Fim da leitura";
        podeLer = false;
    }
    stream.Close();
    bool balanceada = true;
    lblAltura.Text = "Altura : " + Convert.ToString(
        arvore.alturaArvore(arvore.Raiz, ref balanceada));
    chkBalanceada.Checked = balanceada;
}
```

9. Grafos

O estudo de redes tornou-se um dos grandes tópicos científicos do século XXI, embora os matemáticos e outros especialistas venham estudando redes por muitas centenas de anos. Avanços recentes em tecnologia de computação (por exemplo, a Internet) e em teoria social (a rede social, popularmente concebida no conceito de "[seis graus de separação](#)") acabou digirindo holofotes ao estudo de redes.

Neste capítulo, estudaremos as como redes são modeladas com grafos. Definiremos o que é um grafo, como eles podem ser representados e como implementar importantes algoritmos para grafos. Também discutiremos a importância de escolher a representação correta dos dados quando trabalhamos com grafos, pois a eficiência de algoritmos de grafos depende da estrutura de dados usada.

1. Definições de Grafo

Um **grafo** consiste em um conjunto de **vértices** (vertex) e um conjunto de **arestas** (edges). Pense no mapa de seu estado. Cada cidade é conectada com outras cidades através de algum tipo de estrada. Um mapa é um tipo de grafo. Cada cidade é um vértice e a estrada que conecta duas cidades é uma aresta. Arestas são especificadas como um *par* (v_1, v_2), onde v_1 e v_2 são dois vértices no grafo. Uma aresta pode também ter um **peso**, por vezes também conhecido como um **custo**.

Um grafo cujos pares são ordenados (indicam a ordem de percurso de v_1 a v_2) é chamado de um **grafo direcionado** ou de **dígrafo**. Nesse tipo de grafo, as arestas são unidireccionais, ou seja, (u, v) é uma aresta diferente de (v, u) . Um grafo ordenado é ilustrado na figura 1:

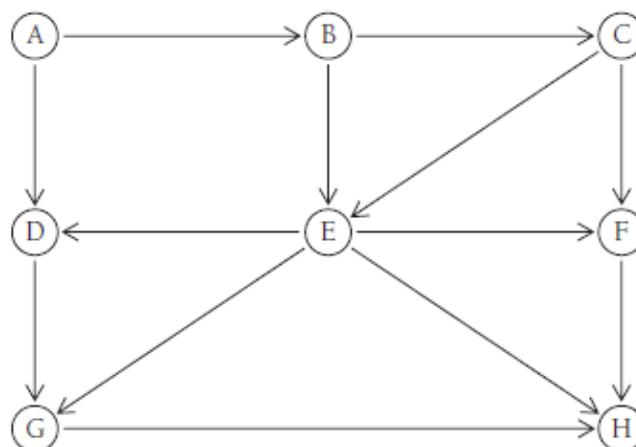


Figura 1 - Um dígrafo (grafo direcionado)

Se um grafo não é ordenado, será chamado de **grafo não-ordenado** ou, simplesmente, de grafo, possuindo, portanto, arestas bidirecionais, onde o par que define uma aresta serve para ida e para volta, ou seja (u, v) é a mesma aresta que (v, u) . Um grafo não-ordenado é ilustrado pela figura 2:

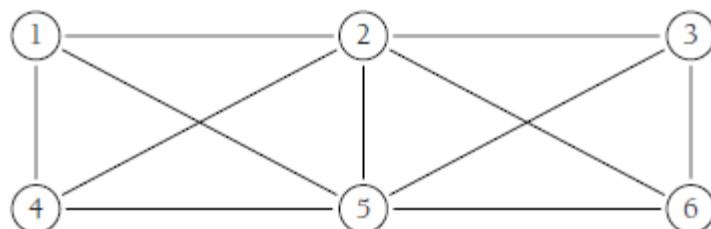
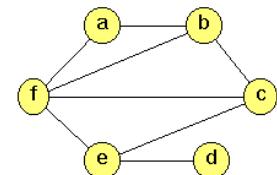


Figura 2 - grafo não-direcionado

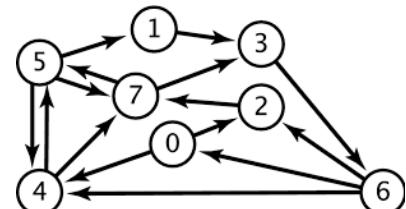
Um **caminho** é uma **sequência de vértices** em um grafo de tal forma que todos os vértices dessa sequência sejam conectados por arestas. O comprimento de um caminho é o número de arestas desde o primeiro vértice até o último vértice. Um caminho pode também consistir de um vértice para si próprio, o que é chamado de **laço**. Laços tem um comprimento de 0.

Um **ciclo** ou **círculo** é um caminho de comprimento de pelo menos 1, num digrafo, em que o vértice de início é também o vértice final. Em um grafo direcionado, as arestas podem ser uma única aresta (apenas ida), mas em um grafo não-direcionado, as arestas devem ser distintas (ida e volta).

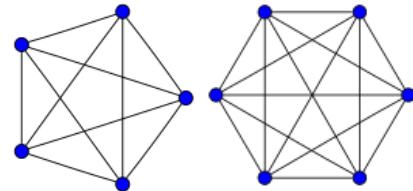
Um grafo não-direcionado é considerado **conectado** se existe um caminho partindo de cada vértice para cada um dos demais vértices.



Em um grafo direcionado, esta condição é chamada de **fortemente conectado**. Um digrafo que não seja fortemente conectado, mas que seja considerado conectado, é chamado de **fracamente conectado**.



Se um grafo tem uma aresta entre cada conjunto de vértices, é chamado de um **grafo completo**.



Sistemas do Mundo Real modelados por Grafos

Grafos são usados para modelar vários diferentes tipos de sistemas do mundo real. Um exemplo é o fluxo de tráfego. As arestas representam as próprias ruas e os vértices representam intersecções de ruas. Arestas com pesos podem ser usadas para representar distâncias, limites de velocidades, número de pistas ou um conjunto de vários fatores para análise. Modeladores podem usar o sistema para determinar as melhores rotas e as ruas que possuem probabilidade de passar por congestionamentos.

Qualquer tipo de sistema de transporte pode ser modelado usando um grafo. Por exemplo, uma empresa aérea pode modelar seu sistema de vôo usando um grafo. Cada aeroporto é um vértice e cada vôo de um vértice a outro seria uma aresta. Uma aresta ponderada pode representar o custo de um vôo de um aeroporto a outro ou, talvez, a distância de um aeroporto a outro, dependendo do que está sendo modelado.

2. A Classe Grafo

Em um primeiro vislumbre, um grafo se parece muito com uma árvore e você poderá se sentir tentado a tentar criar uma classe Grafo como se fosse uma árvore. No entanto, haverá problemas quando se tenta usar uma implementação baseada em referências (ponteiros), de forma que procuraremos um esquema diferente para representar tanto os vértices quanto as arestas.

Representação de Vértices

O primeiro passo que tomaremos para construir a classe Grafo é construir uma classe Vertice, para armazenar os vértices de um grafo. Esta classe tem deveres semelhantes aos das classes NoLista ou NoArvore, usadas nas estruturas de dados Lista Ligada e Árvore Binária.

A classe Vertice precisa de, ao menos, dois membros de dados : um para os dados que identificam o vértice e o outro como um Boolean que usaremos para manter o rastreio de "visitas" ao vértice. Chamaremos esses membros de dados **rotulo** e **foiVisitado**, respectivamente.

O único método que precisaremos para essa classe é um construtor que nos permita configurar os membros rotulo e foiVisitado. Não usaremos um construtor default nesta implementação porque, a cada vez que fizermos a primeira referência a um objeto Vertice, nós já o estaremos instanciando.

Segue abaixo o código da classe Vertice:

```
using System;
namespace Grafos
{
    class Vertice
    {
        public bool foiVisitado;
        public string rotulo;

        public Vertice(string label)
        {
            rotulo = label;
            foiVisitado = false;
        }
    }
}
```

Armazenaremos a lista de vértices em um **vetor** e os referenciaremos na classe Grafo pela sua **posição no vetor**.

Representação de Arestas

A real informação sobre um grafo é armazenada nas arestas, pois elas detalham a estrutura do grafo. Como mencionamos anteriormente, é tentador representar um grafo como uma árvore binária, mas fazer isso seria um erro. Uma árvore binária tem uma representação relativamente fixa, já que um nó ancestral pode somente ter dois nós descendentes, enquanto que a estrutura de um grafo requer muito mais flexibilidade, já que, por exemplo, um único vértice pode ter muitas arestas a ele ligadas ou apenas uma única aresta.

O método que escolheremos para representar as arestas de um grafo é chamado de **Matriz de Adjacência**, uma matriz bidimensional onde os elementos indicam se uma aresta existe entre dois vértices. A figura 3 ilustra como uma matriz de adjacência funciona para o grafo na figura.

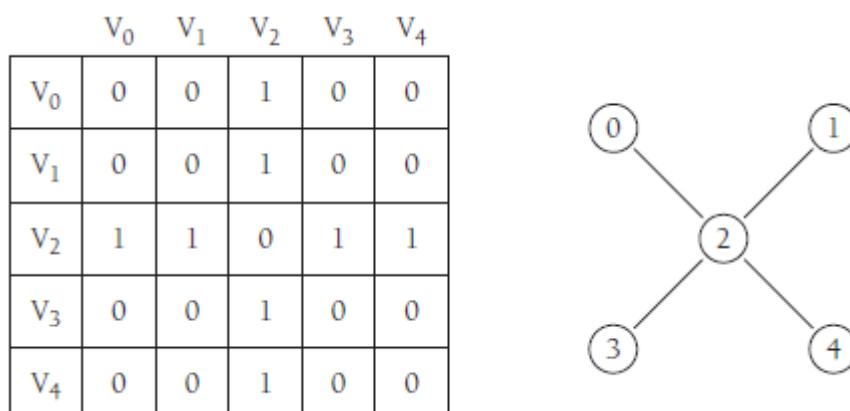


Figura 3

Os vértices são listados como os cabeçalhos para as linhas e as colunas. Se há uma aresta entre dois vértices, o valor 1 é colocado na posição indicada pela coordenada dada pelos vértices. Se não há uma aresta entre eles, essa posição recebe um 0. Obviamente, aqui você também pode usar valores lógicos para indicar a mesma situação. Temos outro exemplo na próxima figura:

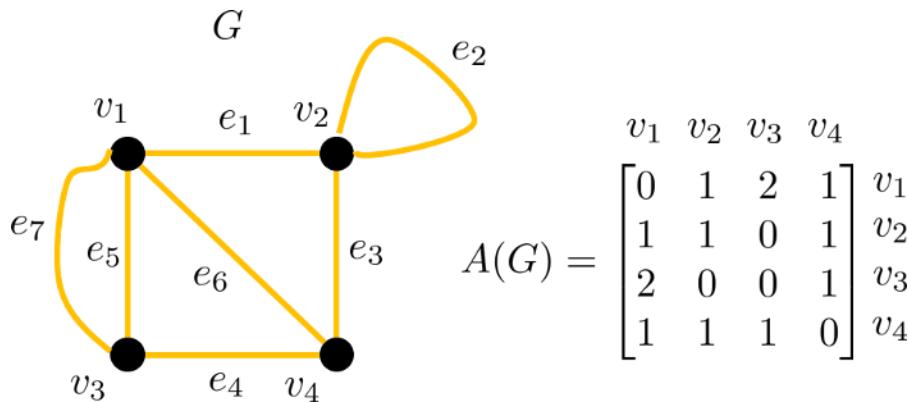


Figura 4

Construção de um Grafo

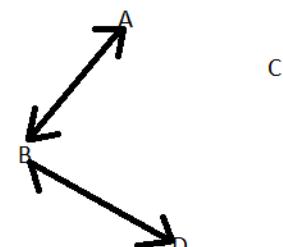
Agora que temos uma forma de representar os vértices e as arestas, estamos prontos para construir um grafo. Primeiramente, precisamos criar uma lista com os vértices do grafo. Aqui está o código para um pequeno grafo com quatro vértices:

```
int nVertices = 0;
vertices[nVertices++] = new Vertice("A"); // 0
vertices[nVertices++] = new Vertice("B"); // 1
vertices[nVertices++] = new Vertice("C"); // 2
vertices[nVertices++] = new Vertice("D"); // 3
```

Agora precisamos adicionar as arestas que conectam os vértices. Segue o código para adicionar duas arestas, supondo que a matriz está toda zerada inicialmente:

```
adjMatrix[0,1] = 1;
adjMatrix[1,0] = 1;
adjMatrix[1,3] = 1;
adjMatrix[3,1] = 1;
```

O código acima estabelece que existe uma aresta entre os vértices A e B, e uma outra aresta existe entre os vértices B e D.



Com essas peças, estamos prontos para uma definição preliminar da classe Grafo :

```
public class Grafo
{
    private const int NUM_VERTICES = 20;
    private Vertice[] vertices;
    private int[,] adjMatrix;
    int numVerts;
    DataGridView dgv; // para exibir a matriz de adjacência num formulário
    public Grafo(DataGridView dgv)
    {
        this.dgv = dgv;
        vertices = new Vertice[NUM_VERTICES];
        adjMatrix = new int[NUM_VERTICES, NUM_VERTICES];
```

```
numVerts = 0;
for (int j = 0; j < NUM_VERTICES; j++)           // zera toda a matriz
    for (int k = 0; k < NUM_VERTICES; k++)
        adjMatrix[j, k] = 0;
}

public void NovoVertice(string label)
{
    vertices[numVerts] = new Vertice(label);
    numVerts++;
    if (dgv != null) // se foi passado como parâmetro um dataGridView para exibição
    {                // se realiza o seu ajuste para a quantidade de vértices
        dgv.RowCount = numVerts+1;
        dgv.ColumnCount = numVerts+1;
        dgv.Columns[numVerts].Width = 45;
    }
}

public void NovaAresta(int start, int eend)
{
    adjMatrix[start, eend] = 1; // adjMatrix[eend, start] = 1; ISSO GERA CICLOS!!!
}

public void ExibirVertice(int v)
{
    Console.WriteLine(vertices[v].rotulo + " ");
}

public void ExibirVertice(int v, TextBox txt)
{
    txt.Text += vertices[v].rotulo + " ";
}
}
```

O construtor redimensiona o vetor de vértices e a matriz de adjacência para o número especificado na constante NUM_VERTICES. O membro de dados numVerts armazena a quantidade atual na lista de vértices, de maneira que inicialmente ele vale zero. Finalmente, a matriz de adjacência é iniciada pela atribuição de zero a todos os seus elementos.

O método NovoVertice recebe como parâmetro uma string para o rótulo do vértice, instancia um novo vértice, adiciona-o ao vetor de vértices e ajusta o valor de numVerts. O método NovaAresta recebe como parâmetros dois inteiros, que representam dois vértices e indicam que uma aresta existe entre eles. Finalmente, o método ExibirVertice exibe o rótulo do vértice especificado.

3. Uma primeira aplicação de Grafos : Ordenação Topológica

Ordenação Topológica envolve exibir a ordem específica em que uma sequência de vértices deve ser seguida em um grafo dirigido. A sequência de disciplinas que um estudante de um curso técnico deve se matricular no seu itinerário para sua formação pode ser modelada como um grafo dirigido. Um aluno não pode se matricular na disciplina Estruturas de Dados sem que ele tenha realizado as duas disciplinas introdutórias de programação, por exemplo.

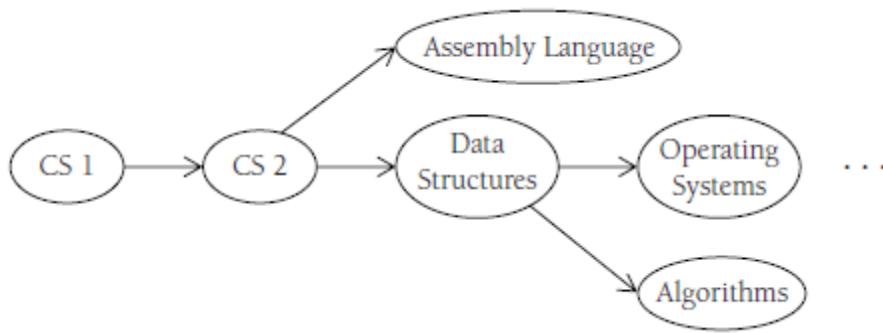


Figura 5

A figura 5 ilustra uma parte de um modelo de grafo dirigido de um Currículo típico do curso Ciência da Computação.

Uma ordenação topológica desse grafo resultaria na seguinte sequência:

1. CS1
2. CS2
3. Assembly Language
4. Data Structures
5. Operating Systems
6. Algorithms

As disciplinas 3 e 4 podem ser seguidas ao mesmo tempo, da mesma maneira que as disciplinas 5 e 6 mas estas duas dependem da disciplina 4 para poderem ser realizadas.

Um Algoritmo para Ordenação Topológica

O algoritmo básico para a ordenação topológica é bastante simples:

1. Encontre um vértice que não tenha sucessores.
2. Adicione esse vértice a uma lista de vértices.
3. Remova o vértice do grafo.
4. Repita os passos 1 a 3 até que todos os vértices sejam removidos.

Claramente, o desafio está nos detalhes da implementação mas este é o básico da ordenação topológica.

O algoritmo na realidade trabalhará **do fim** do grafo dirigido **para o seu começo**. Observe novamente a figura 4. Assumindo que Operating Systems e Algorithms são os últimos vértices (ignore as reticências), nenhum deles tem sucessores e, portanto, devem ser adicionados à lista e removidos do grafo. Em seguida vem Assembly Language e Data Structures. Esses vértices agora não possuem sucessores e, portanto, devem ser removidos do grafo e adicionados à lista. Em seguida, virá CS2. Seus sucessores foram removidos de forma que ele deve ser adicionado à lista e removido do grafo. Finalmente, sobra apenas a disciplina CS1.

Implementação do Algoritmo

Precisaremos de dois métodos para a ordenação topológica— um método para **determinar se um vértice não tem sucessores** e um método para **remover um vértice do grafo**.

Primeiramente vamos estudar o método para determinar que um vértice não possui sucessores. Um vértice sem sucessores será encontrado na matriz de adjacência em uma linha onde todas as colunas sejam zeros. Nossa método usará loops for aninhados para verificar cada conjunto de colunas, linha por linha. Se o valor 1 é encontrado em uma coluna, então o loop interno é deixado

e a próxima linha é verificada. Se uma linha com apenas zeros em suas colunas é encontrada, então o número dessa linha é retornado pelo método. Se os dois loops são completos e nenhum número de linha é retornado, então o valor -1 é retornado, indicando que não há vértices sem sucessores. A seguir temos o código:

```
public int SemSucessores() // encontra e retorna a linha de um vértice sem sucessores
{
    bool temAresta;
    for (int linha = 0; linha < numVerts; linha++)
    {
        temAresta = false;
        for (int col = 0; col < numVerts; col++)
            if (adjMatrix[linha, col] > 0)
            {
                temAresta= true;
                break;
            }
        if (!temAresta)
            return linha;
    }
    return -1;
}
```

Em seguida precisamos ver como remover um vértice do grafo. A primeira coisa que temos que fazer é remover o vértice da lista de vértices. Isto é fácil. A seguir, precisamos remover a linha e coluna da matriz de adjacência, seguido pela movimentação das linhas e das colunas acima e à direita do vértice : elas devem ser movidas para abaixo e para a esquerda para preencher o espaço vazio deixado pelo vértice removido.

Para realizar essa operação, escreveremos um método chamado `removerVertice`, que incluirá dois métodos auxiliares, `moverLinha` e `moverColuna`. Aqui temos o código:

```
public void RemoverVertice(int vert)
{
    if (dgv != null)
    {
        MessageBox.Show($"Matriz de Adjacências antes de remover vértice {vert}");
        ExibirAdjacencias();
    }

    if (vert != numVerts-1)
    {
        for (int j = vert; j < numVerts-1; j++)// remove vértice do vetor
            vertices[j] = vertices[j+1];

        // remove vértice da matriz
        for (int row = vert; row < numVerts; row++)
            MoverLinhas(row, numVerts-1);
        for (int col = vert; col < numVerts; col++)
            MoverColunas(col, numVerts-1);
    }
    numVerts--;
    if (dgv != null)
    {
        MessageBox.Show($"Matriz de Adjacências após remover vértice {vert}");
        ExibirAdjacencias();
        MessageBox.Show("Retornando à ordenação");
    }
}
```

```
private void MoverLinhas(int row, int length)
{
    if (row != numVerts-1)
        for (int col = 0; col < length; col++)
            adjMatrix[row, col] = adjMatrix[row+1, col]; // desloca para excluir
}

private void MoverColunas(int col, int length)
{
    if (col != numVerts - 1)
        for (int row = 0; row < length; row++)
            adjMatrix[row, col] = adjMatrix[row, col+1]; // desloca para excluir
}

public void ExibirAdjacencias()
{
    dgv.RowCount = numVerts+1;
    dgv.ColumnCount = numVerts+1;
    for (int j = 0; j < numVerts; j++)
    {
        dgv.Rows[j + 1].Cells[0].Value = vertices[j].rotulo;
        dgv.Rows[0].Cells[j+1].Value = vertices[j].rotulo;
        for (int k = 0; k < numVerts; k++)
            dgv.Rows[j + 1].Cells[k + 1].Value = Convert.ToString(adjMatrix[j, k]);
    }
}
```

Agora precisamos de um método para controlar o processo de ordenação:

```
public String OrdenacaoTopologica()
{
    Stack<String> gPilha = new Stack<String> (); //guarda a sequência de vértices
    int origVerts = numVerts;
    while (numVerts > 0)
    {
        int currVertex = SemSucessores();
        if (currVertex == -1)
            return "Erro: grafo possui ciclos.";
        gPilha.Push(vertices[currVertex].rotulo);      // empilha vértice
        RemoverVertice(currVertex);
    }
    String resultado = "Sequência da Ordenação Topológica: ";
    while (gPilha.Count > 0)
        resultado+= gPilha.Pop() + " ";      // desempilha para exibir
    return resultado;
}
```

O método OrdenacaoTopologica() percorre os vértices do grafo, encontrando um vértice sem sucessores, removendo-o e então, indo para o próximo vértice.

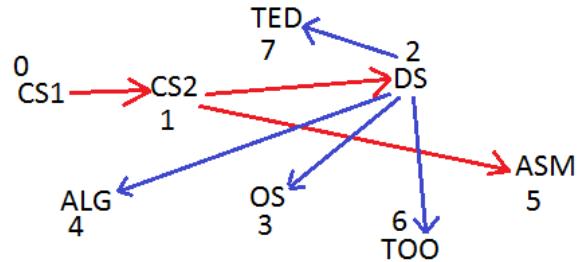
Cada vez que um vértice é removido, seu rótulo é empilhado na pilha gStack. Uma pilha é uma estrutura de dados conveniente para usarmos porque o primeiro vértice encontrado é, realmente, o último (ou um dos últimos) vértices no grafo. Quando o método TopSort termina, o conteúdo da pilha terá o último vértice empilhado no fundo da pilha e o primeiro vértice estará no topo da pilha.

Nós meramente temos de desempilhar todos os elementos da pilha para exibir a ordem topológica correta do grafo.

Esses são todos os métodos que precisamos ter, da classe Grafo, para realizar uma ordenação topológica em um grafo dirigido.

A seguir temos um tratador de evento de botão que testa nossa implementação:

```
private void btnTeste1_Click(object sender, EventArgs e)
{
    Grafo oGrafo = new Grafo(dgvGrafo);
    oGrafo.NovoVertice("CS1"); // 0
    oGrafo.NovoVertice("CS2"); // 1
    oGrafo.NovoVertice("DS"); // 2
    oGrafo.NovoVertice("OS"); // 3
    oGrafo.NovoVertice("ALG"); // 4
    oGrafo.NovoVertice("ASM"); // 5
    oGrafo.NovoVertice("TOO"); // 6
    oGrafo.NovoVertice("TED"); // 7
    oGrafo.NovaAresta(0, 1);
    oGrafo.NovaAresta(1, 2);
    oGrafo.NovaAresta(1, 5);
    oGrafo.NovaAresta(2, 3);
    oGrafo.NovaAresta(2, 4);
    oGrafo.NovaAresta(2, 6);
    oGrafo.NovaAresta(2, 7);
    txtSaida.Text = oGrafo.OrdenacaoTopologica();
}
```



Na próxima figura vemos uma sugestão para o formulário do aplicativo que usaremos para testar a ordenação topológica:



Ordenação Topológica

Teste 1

	CS1	CS2	DS	OS	ALG	ASM	TOO	TED
CS1	0	1	0	0	0	0	0	0
CS2	0	0	1	0	0	1	0	0
DS	0	0	0	1	1	0	1	1
OS	0	0	0	0	0	0	0	0
ALG	0	0	0	0	0	0	0	0
ASM	0	0	0	0	0	0	0	0
TOO	0	0	0	0	0	0	0	0
*	TED	0	0	0	0	0	0	0

Matriz de Adjacências apóis remover vértice 3

OK

Ordenação Topológica

Teste 1

	CS1	CS2	DS	ALG	ASM	TOO	TED
CS1	0	1	0	0	0	0	0
CS2	0	0	1	0	1	0	0
DS	0	0	0	1	0	1	1
ALG	0	0	0	0	0	0	0
ASM	0	0	0	0	0	0	0
TOO	0	0	0	0	0	0	0
*	TED	0	0	0	0	0	0

Retornando à ordenação

OK

Ordenação Topológica

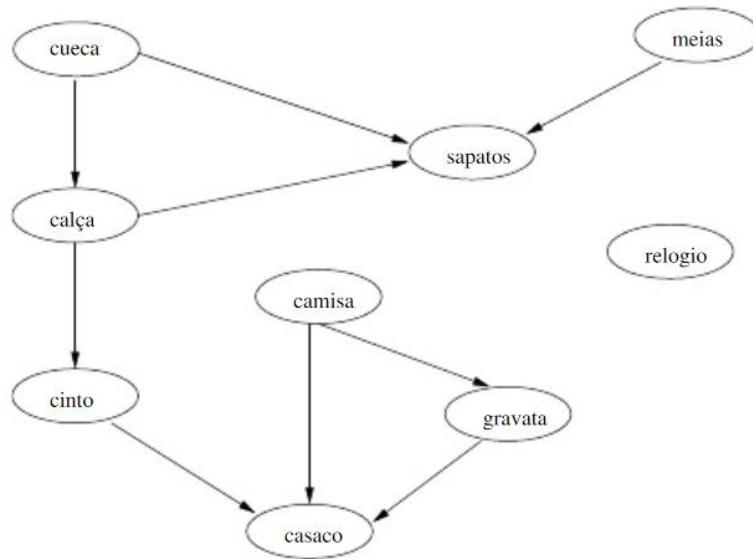
Teste 1

Seqüência da Ordenação Topológica: CS1 CS2 DS TED TOO ASM ALG OS

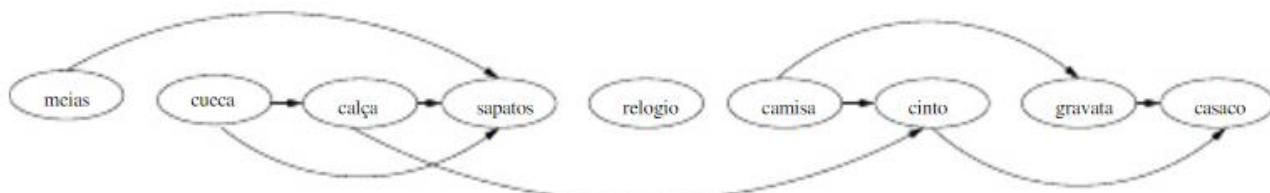
*	CS1
---	-----

Um outro exemplo:

Vestir-se pela manhã : suponha que sua sonolência impede seu processamento cerebral pela manhã e você está cansado de sair de casa vestindo as meias sobre os sapatos. Então, você resolve que deve fazer uma lista do que deverá vestir e em que ordem. Primeiramente, você monta um grafo com as roupas e quais dependem de quais:



Efetuando-se a ordenação topológica, conforme o algoritmo apresentado, teremos a seguinte lista (pode haver variações, já que há mais de um vértice de entrada):



Codifique, no formulário, o evento Click de um novo botão, para efetuar o teste desse grafo.

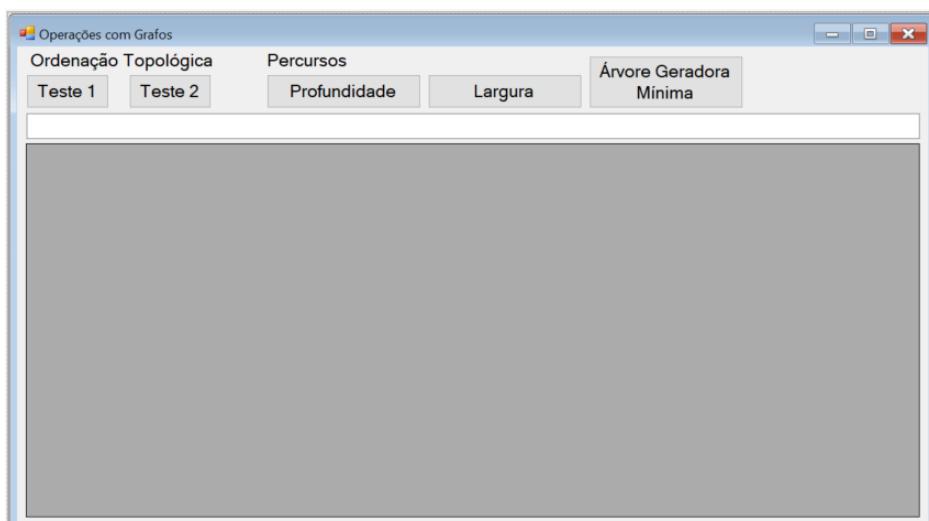
4. Percursos em Grafos

A determinação de quais vértices podem ser alcançados a partir de um vértice especificado é uma atividade comum realizada em grafos. Nós podemos querer saber quais estradas levam de uma cidade para outras cidades no mapa ou quais vôos podem nos levar de um aeroporto a outros.

Essas operações são realizadas em um grafo usando um algoritmo de busca. Existem duas buscas fundamentais que nós podemos realizar num grafo: a busca em profundidade e a busca em largura.

Nesta seção, examinaremos cada um desses algoritmos.

Primeiramente, daremos sequência em nosso projeto Windows Forms, modificando-o como vemos ao lado:



Percorso em Profundidade

Percorso em profundidade envolve o percurso de um caminho desde o vértice inicial até que se encontre o último vértice e, então, retornar um passo e seguir o próximo caminho até que ele encontre o último vértice e assim por diante até que não restem mais caminhos. Um diagrama da pesquisa em profundidade é mostrado na Figura 6.

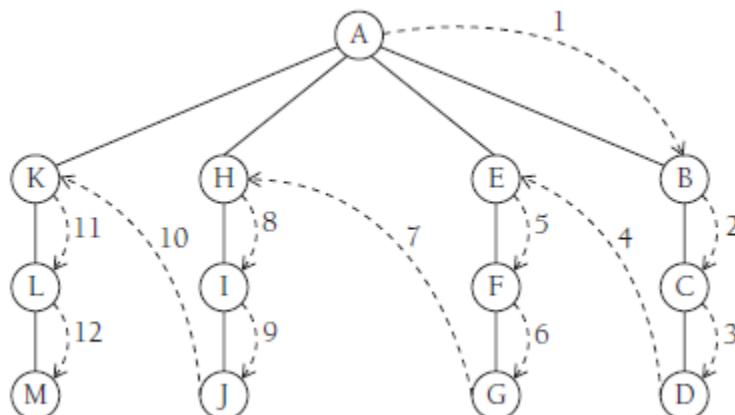


Figura 6 - Percorso em profundidade

Num alto nível de abstração, o algoritmo de percurso em profundidade funciona da seguinte maneira: primeiramente, escolha um ponto de início, que pode ser qualquer vértice. Visite esse vértice, empilhe-o numa pilha e o marque como visitado. Então vá para o próximo vértice que ainda não foi visitado, empilhe-o e o marque como visitado. Isto continua até que se chegue ao último vértice.

Então você verifica se o vértice do topo tem algum vértice adjacente ainda não visitado. Se ele não tem, então você o desempilha e verifica o próximo vértice. Se você achar um, você deve começar a visitar os vértices adjacentes até que não haja mais nenhum, verifique se há mais vértices adjacentes não visitados e continue o processo. Quando você finalmente encontrar o último vértice na pilha e não houver mais nenhum vértice adjacente não visitado, você terminou o percurso em profundidade.

O primeiro trecho de código que devemos desenvolver é um método para obter um vértice adjacente não visitado. Nossa código deve primeiramente ir até a linha para o vértice especificado e determinar se o valor 1 está armazenado em uma de suas colunas. Se estiver, então existe um vértice adjacente. Podemos então, facilmente, verificar se o vértice foi visitado ou não. Eis o código para esse método:

```
private int ObterVerticeAdjacenteNaoVisitado(int v)
{
    for (int j = 0; j <= numVerts - 1; j++)
        if ((adjMatrix[v, j] == 1) && (!vertices[j].foiVisitado))
            return j;
    return -1;
}
```

Agora vamos estudar o método que realiza o percurso em profundidade:

```
public void PercorsoEmProfundidade(TextBox txt)
{
    txt.Clear();
    Stack<int> gPilha = new Stack<int>(); // para guardar a sequência de vértices
    vertices[0].foiVisitado = true;
    ExibirVertice(0, txt);
    gPilha.Push(0);
```

```
int v;
while (gPilha.Count > 0)
{
    v = ObterVerticeAdjacenteNaoVisitado(gPilha.Peek());
    if (v == -1)
        gPilha.Pop();
    else
    {
        vertices[v].foiVisitado = true;
        ExibirVertice(v, txt);
        gPilha.Push(v);
    }
}
for (int j = 0; j <= numVerts - 1; j++)
    vertices[j].foiVisitado = false;
```

}

O evento Click do btnProfundidade abaixo realiza um percurso em profundidade no grafo exbido na Figura 5:

```
private void BtnProfundidade_Click(object sender, EventArgs e)
{
    Grafo aGraph = new Grafo(dgvGrafo);
    aGraph.NovoVertice("A");
    aGraph.NovoVertice("B");
    aGraph.NovoVertice("C");
    aGraph.NovoVertice("D");
    aGraph.NovoVertice("E");
    aGraph.NovoVertice("F");
    aGraph.NovoVertice("G");
    aGraph.NovoVertice("H");
    aGraph.NovoVertice("I");
    aGraph.NovoVertice("J");
    aGraph.NovoVertice("K");
    aGraph.NovoVertice("L");
    aGraph.NovoVertice("M");
    aGraph.NovaAresta(0, 1);
    aGraph.NovaAresta(1, 2);
    aGraph.NovaAresta(2, 3);
    aGraph.NovaAresta(0, 4);
    aGraph.NovaAresta(4, 5);
    aGraph.NovaAresta(5, 6);
    aGraph.NovaAresta(0, 7);
    aGraph.NovaAresta(7, 8);
    aGraph.NovaAresta(8, 9);
    aGraph.NovaAresta(0, 10);
    aGraph.NovaAresta(10, 11);
    aGraph.NovaAresta(11, 12);
    aGraph.exibirAdjacencias();
    aGraph.PercorsoEmProfundidade(txtSaida);
}
```

A saída deste programa é:

Eis a versão recursiva desse método:

```
void ProcessarNo(int i)
{
    Console.Write(vertices[i].rotulo);
}

public void PercursoEmProfundidadeRec(int[,] adjMatrix, int numverts, int part)
{
    int i;
    ProcessarNo(part);
    vertices[part].foiVisitado = true;
    for (i = 0; i < numverts; ++i)
        if (adjMatrix[part, i] == 1 && !vertices[i].foiVisitado)
            PercursoEmProfundidadeRec(adjMatrix, numverts, i);
}
```

Usando os métodos anteriores como modelo, implemente a versão recursiva desse percurso, após modificar esses métodos para que eles usem os componentes do formulário para exibir os resultados, ao invés do Console. Crie um botão para disparar o evento que executa esse percurso recursivo.

Percorso em Largura

O Percurso em Largura é iniciado em um primeiro vértice e tenta visitar os vértices tão próximos quanto possível ao primeiro vértice. Em essência, este percurso faz um movimento através de um grafo camada por camada, examinando as camadas mais próximas ao primeiro vértice e movendo-se até as camadas mais distantes do vértice inicial. A figura 7 mostra como o percurso em largura funciona:

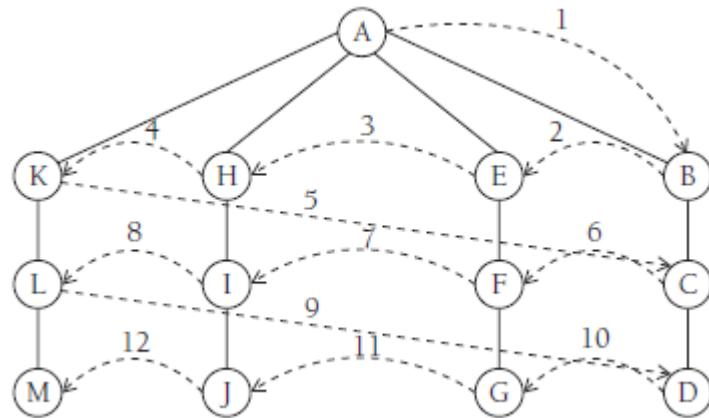
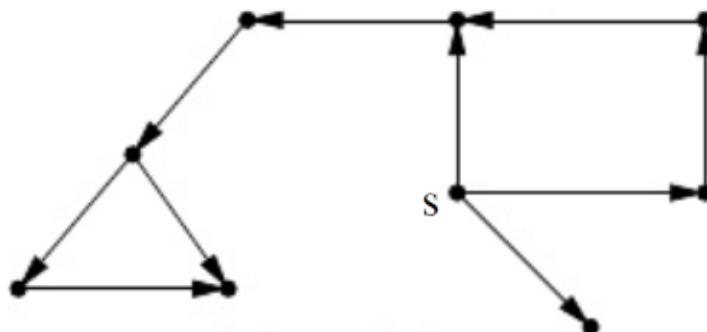
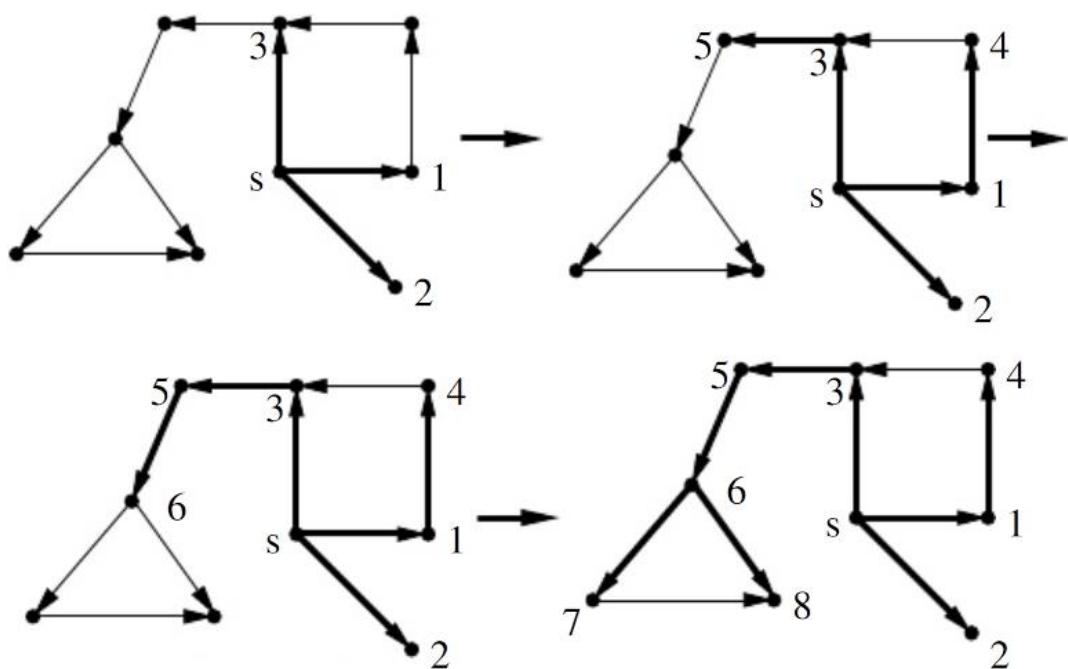


Figura 7

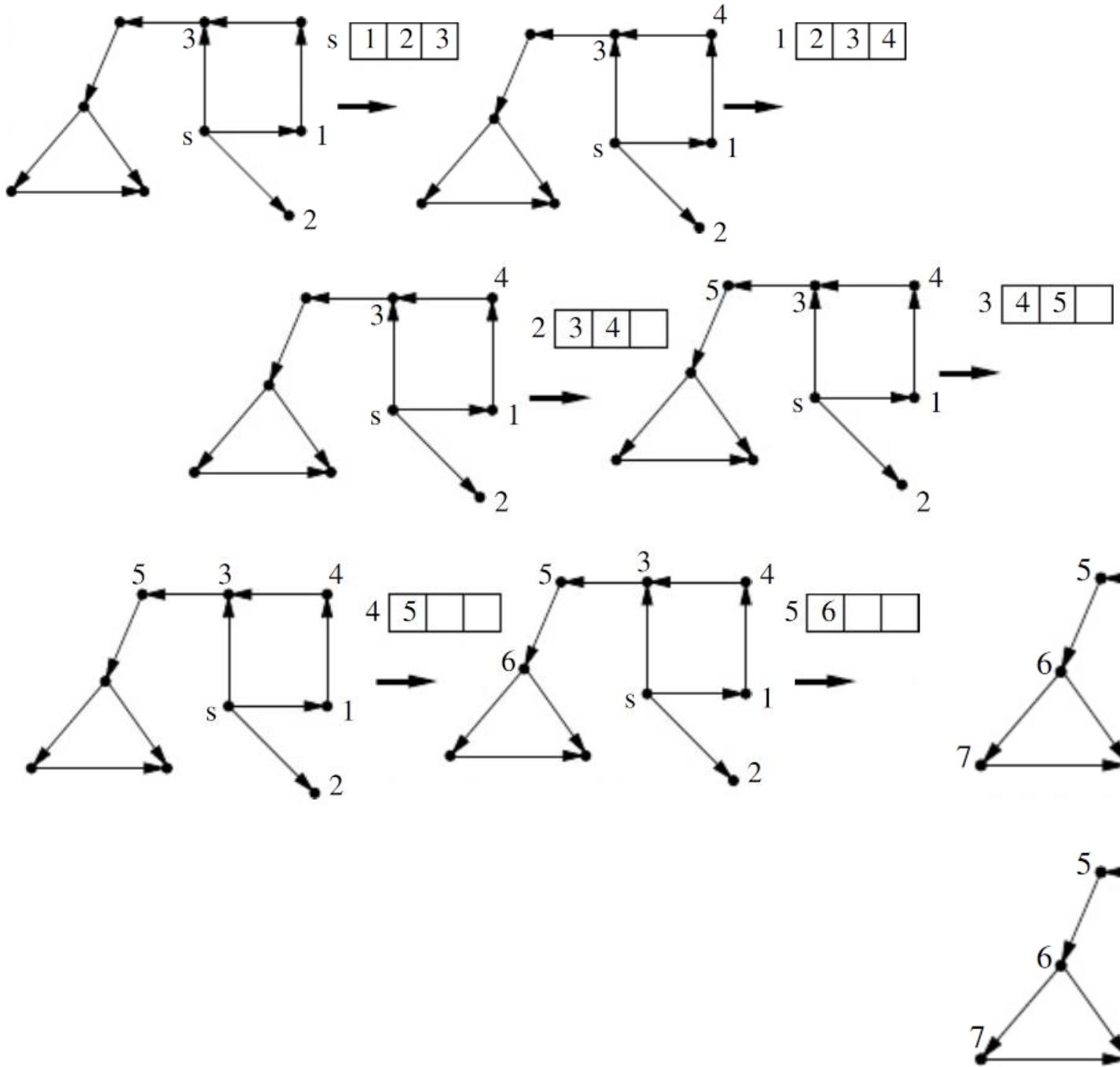
A idéia do percurso em largura (ou em amplitude) é bastante simples: os vértices do grafo são visitados nível a nível, ou seja, todos os vértices a uma distância k do vértice inicial são visitados antes de qualquer vértice a uma distância $k+1$ do inicial. O exemplo abaixo é baseado em <http://pt.scribd.com/doc/51625528/10/BFS-Breadth-First-Search-Percorso-em-Largura>, do professor Pedro Jussieu de Rezende, do Instituto de Computação da UNICAMP. O vértice inicial foi rotulado S:



Começamos, então, a visita a todos os vértices adjacentes a S e assim por diante (note que não há backtracking).



O algoritmo de percurso por largura usa uma fila ao invés de uma pilha, embora uma pilha também possa ser usada. Observe as figuras abaixo para entender o uso da fila:



O algoritmo segue abaixo:

1. Encontre um vértice não visitado que seja adjacente ao vértice atual, marque-o como visitado, e o adicione a uma fila.
2. Se um vértice adjacente e não visitado não pode ser encontrado, , remova um vértice da fila (desde que haja um vértice na fila para ser removido), torne-o o vértice atual e recomece.
3. Se o segundo passo não pode ser realizado porque a fila está vazia, então o algoritmo terminou.

Vejamos o código do algorítmo:

```
public void percursoPorLargura(TextBox txt)
{
    txt.Clear();
    Queue<int> gQueue = new Queue<int>();
    vertices[0].foiVisitado = true;
    ExibirVertice(0, txt);
    gQueue.Enqueue(0);
    int vert1, vert2;
    while (gQueue.Count > 0 )
    {
        vert1 = gQueue.Dequeue();
        vert2 = ObterVerticeAdjacenteNaoVisitado(vert1);
        while (vert2 != -1)
        {
            vertices[vert2].foiVisitado = true;
            ExibirVertice(vert2, txt);
            gQueue.Enqueue(vert2);
            vert2 = ObterVerticeAdjacenteNaoVisitado(vert1);
        }
    }
    for (int i = 0; i < numVerts; i++)
        vertices[i].foiVisitado = false;
}
```

Observe que há duas repetições neste método. A repetição externa é executada enquanto a fila tem dados dentro dela, e a repetição interna verifica vértices adjacentes para verificar se eles já foram visitados. A repetição **for** ao final do método simplesmente limpa os vértices para poderem ser usados por outros métodos.

O evento Click do btnProfundidade abaixo testa este código, usando o grafo da figura 6:

```
private void BtnLargura_Click(object sender, EventArgs e)
{
    Grafo aGraph = new Grafo(dgvGrafo);
    aGraph.NovoVertice("A");
    aGraph.NovoVertice("B");
    aGraph.NovoVertice("C");
    aGraph.NovoVertice("D");
    aGraph.NovoVertice("E");
    aGraph.NovoVertice("F");
    aGraph.NovoVertice("G");
    aGraph.NovoVertice("H");
    aGraph.NovoVertice("I");
    aGraph.NovoVertice("J");
    aGraph.NovoVertice("K");
    aGraph.NovoVertice("L");
    aGraph.NovoVertice("M");
    aGraph.NovaAresta(0, 1);
    aGraph.NovaAresta(1, 2);
    aGraph.NovaAresta(2, 3);
    aGraph.NovaAresta(0, 4);
    aGraph.NovaAresta(4, 5);
    aGraph.NovaAresta(5, 6);
    aGraph.NovaAresta(0, 7);
    aGraph.NovaAresta(7, 8);
    aGraph.NovaAresta(8, 9);
    aGraph.NovaAresta(0, 10);
    aGraph.NovaAresta(10, 11);
    aGraph.NovaAresta(11, 12);
    aGraph.exibirAdjacencias();
    aGraph.percursoPorLargura(txtSaida);
}
```

A saída dessa operação é:

		Operações com Grafos														
		Ordenação Topológica							Percursos							
		Teste 1		Teste 2		Profundidade			Largura					Árvore Geradora Mínima		
A B E H K C F I L D G J M																
▶		A	B	C	D	E	F	G	H	I	J	K	L	M		
	A	0	1	0	0	1	0	0	1	0	0	1	0	0		
	B	0	0	1	0	0	0	0	0	0	0	0	0	0		
	C	0	0	0	1	0	0	0	0	0	0	0	0	0		
	D	0	0	0	0	0	0	0	0	0	0	0	0	0		
	E	0	0	0	0	0	1	0	0	0	0	0	0	0		
	F	0	0	0	0	0	0	1	0	0	0	0	0	0		
	G	0	0	0	0	0	0	0	0	0	0	0	0	0		
	H	0	0	0	0	0	0	0	1	0	0	0	0	0		
	I	0	0	0	0	0	0	0	0	1	0	0	0	0		
	J	0	0	0	0	0	0	0	0	0	0	0	0	0		
	K	0	0	0	0	0	0	0	0	0	0	1	0	0		
	L	0	0	0	0	0	0	0	0	0	0	0	0	1		
*	M	0	0	0	0	0	0	0	0	0	0	0	0	0		

http://algol.dcc.ufla.br/~heitor/Projetos/TBC_AED_GRAFOS_WEB/TBC_AED_GRAFOS_WEB.html

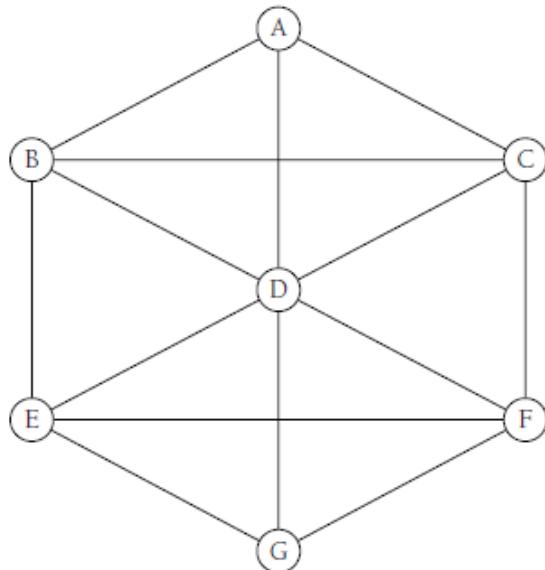
Árvore Geradora Mínima

Quando uma rede é projetada pela primeira vez, é possível haver mais que o número mínimo de conexões entre os nós da rede. As conexões extra são um desperdício de recursos e, se possível, deveriam ser eliminadas. As conexões extra também tornam a rede desnecessariamente complexa para outros estudá-la e entendê-la. O que desejamos é uma rede que contenha apenas o número mínimo de conexões necessárias para conectar os nós. Tal rede, quando aplicada em um grafo, é chamada de **árvore geradora mínima**.

Uma árvore geradora mínima é assim chamada porque ela é construída a partir do número mínimo de arestas necessárias para cobrir cada vértice (abrangência, spanning) e está em forma de árvore porque o grafo resultante é acíclico (não forma ciclos). Existe um ponto importante para ser sempre lembrado: um grafo pode conter várias árvores geradoras mínimas; a árvore geradora mínima criada depende diretamente do vértice inicial do percurso.

Um algoritmo de Árvore Geradora Mínima (Minimum Spanning Tree)

A Figura 8 exibe um grafo para o qual queremos construir a árvore geradora mínima.



Este algoritmo é, na realidade, apenas um algoritmo de percurso em grafo (tanto o de profundidade ou de largura) com o componente adicional de registrar cada aresta que é percorrida. O código também é similar, como vemos abaixo:

Figura 8 - Grafo para gerarmos a árvore geradora mínima

```

public void ArvoreGeradoraMinima(int primeiro, TextBox txt)
{
    txt.Clear();
    Stack<int> gPilha = new Stack<int>(); // para guardar a sequência de vértices
    vertices[primeiro].foiVisitado = true;
    gPilha.Push(primeiro);
    int currVertex, ver;
    while (gPilha.Count > 0)
    {
        currVertex = gPilha.Peek();
        ver = ObterVerticeAdjacenteNaoVisitado(currVertex);
        if (ver == -1)
            gPilha.Pop();
        else
        {
            vertices[ver].foiVisitado = true;
            gPilha.Push(ver);
            ExibirVertice(currVertex, txt);
            txt.Text += "-->";
            ExibirVertice(ver, txt);
            txt.Text += " ";
        }
    }
    for (int j = 0; j <= numVerts - 1; j++)
        vertices[j].foiVisitado = false;
}
  
```

Se você comparar este método com o método para percurso em profundidade, notará que o vértice atual é registrado através da chamada ao método `ExibirVertice` com o vértice atual como argumento. Chamando este método duas vezes, como mostrado no código acima, e a concatenação de caracteres indicando setas, cria a apresentação das arestas que definem a árvore geradora mínima.

Abaixo temos o código do evento Click do `btnArvoreGeradoraMinima`, que cria o grafo da figura 7 e produz a árvore geradora mínima:

```
private void BtnArvoreGeradoraMinima_Click(object sender, EventArgs e)
{
    Grafo aGraph = new Grafo(dgvGrafo);
    aGraph.NovoVertice("A");
    aGraph.NovoVertice("B");
    aGraph.NovoVertice("C");
    aGraph.NovoVertice("D");
    aGraph.NovoVertice("E");
    aGraph.NovoVertice("F");
    aGraph.NovoVertice("G");
    aGraph.NovaAresta(0, 1);
    aGraph.NovaAresta(0, 2);
    aGraph.NovaAresta(0, 3);
    aGraph.NovaAresta(1, 2);
    aGraph.NovaAresta(1, 3);
    aGraph.NovaAresta(1, 4);
    aGraph.NovaAresta(2, 3);
    aGraph.NovaAresta(2, 5);
    aGraph.NovaAresta(3, 5);
    aGraph.NovaAresta(3, 4);
    aGraph.NovaAresta(3, 6);
    aGraph.NovaAresta(4, 5);
    aGraph.NovaAresta(4, 6);
    aGraph.NovaAresta(5, 6);
    aGraph.exibirAdjacencias();
    aGraph.ArvoreGeradoraMinima(0, txtSaida);
}
```

A saída dessa operação vem ao lado:



The screenshot shows a software interface for graph operations. The menu bar includes 'Arquivo', 'Operações com Grafos', 'Ajuda', and 'Sobre'. Below the menu is a toolbar with icons for 'Novo', 'Abrir', 'Salvar', 'Novo Vertice', 'Novo Aresta', 'Exibir Adjacencias', 'Arvore Geradora Minima', and 'Sair'. The main window has tabs: 'Teste 1' (selected), 'Teste 2', 'Profundidade', 'Largura', and 'Árvore Geradora Mínima'. Below the tabs is a command line: 'A -->B B -->C C -->D D -->E E -->F F -->G'. The central part of the window is a 7x7 matrix representing the graph's adjacency matrix:

		A	B	C	D	E	F	G
A	0	1	1	1	0	0	0	
B	0	0	1	1	1	0	0	
C	0	0	0	1	0	1	0	
D	0	0	0	0	1	1	1	
E	0	0	0	0	0	1	1	
F	0	0	0	0	0	0	1	
*	G	0	0	0	0	0	0	0

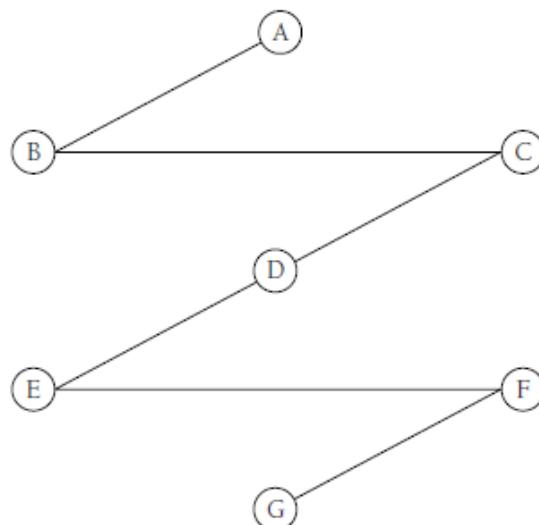


Figura 9 - Árvore geradora

mínima para o grafo da figura

5. Encontrando o Menor Caminho

Uma das operações mais comuns realizadas com grafos é encontrar o menor caminho entre um vértice e outro.

Por exemplo, durante suas férias você viajará para as 10 principais cidades da liga da basebol para ver os jogos em um período de duas semanas. Você quer minimizar o número de milhas que você terá de dirigir para visitar todas as 10 cidades usando um algoritmo de menor caminho.

Outro exemplo e problema de menor caminho é criar uma rede de computadores, onde o custo seria o tempo para transmitir dados entre dois computadores ou o custo de estabelecer e manter a conexão. Um algoritmo de menor caminho pode determinar a maneira mais efetiva para você construir a rede.

Grafos Ponderados

Mencionamos grafos ponderados no início do capítulo. Cada aresta no grafo tem um peso associado, ou custo. Um grafo ponderado é mostrado na figura 10. Grafos ponderados podem ter pesos negativos, mas nos limitaremos nossa discussão a pesos positivos. Também manteremos o foco em grafos dirigidos.

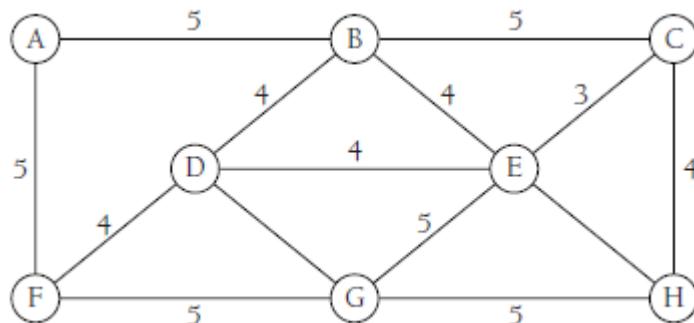


Figura 10 – Um grafo ponderado

Algoritmo de Dijkstra para Determinar o Menor Caminho

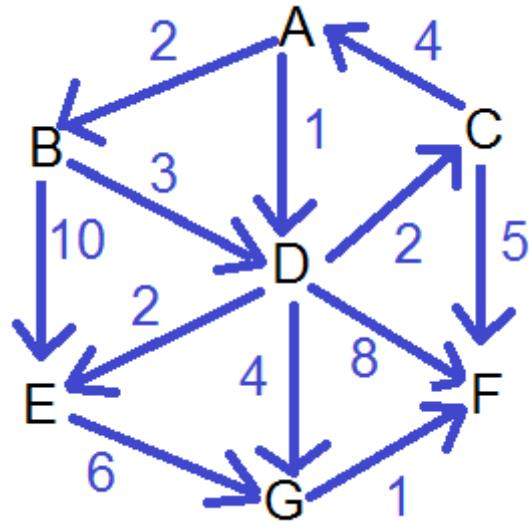
Um dos algoritmos mais famosos na Ciência da Computação é o *Algoritmo de Dijkstra*, que serve para determinar o menor caminho em um grafo ponderado. Recebe seu nome do finado cientista da computação Edsger Dijkstra, que descobriu o algoritmo no final dos anos da década de 1950.

O algoritmo de Dijkstra encontra o comprimento do menor caminho a partir de um vértice especificado **até todos** os outros vértices no grafo. Faz isso usando o que é, geralmente, chamado de estratégia ou algoritmo **gananciosos**. Um algoritmo ganancioso quebra um problema em pedaços, ou estágios, determinando a melhor solução a cada estágio, com cada subsolução contribuindo para a solução final. Um exemplo clássico de algoritmo ganancioso é fazer troco com moedas. Por exemplo, se você comprar algo num mercado por 74 centavos usando uma nota de um real, se o caixa estiver usando um algoritmo ganancioso e quer minimizar o número de moedas retornado, retornará a você uma moeda de 25 centavos e uma de 1 centavo. É claro que há outras soluções para fazer o troco de 26 centavos, mas a solução ótima é a descrita acima.

Usamos o algoritmo de Dijkstra criando uma **tabela para armazenar distâncias conhecidas** do vértice inicial para os outros vértices do grafo. Cada vértice adjacente ao vértice original é visitado, e a tabela é atualizada com informações sobre o peso de cada aresta adjacente. Se uma distância entre dois vértices é conhecida, mas uma distância menor é descoberta ao visitar-se outro vértice, essa informação é alterada na tabela. A tabela também é atualizada para indicar qual vértice leva ao menor caminho.

Imaginemos que o grafo que estamos processando é o representado pela figura ao lado. A sua criação seria semelhante a:

```
Grafo oGrafo = new Grafo();
oGrafo.NovoVertice("A"); // vértice 0
oGrafo.NovoVertice("B"); // vértice 1
oGrafo.NovoVertice("C"); // vértice 2
oGrafo.NovoVertice("D"); // vértice 3
oGrafo.NovoVertice("E"); // vértice 4
oGrafo.NovoVertice("F"); // vértice 5
oGrafo.NovoVertice("G"); // vértice 6
oGrafo.NovaAresta(0, 1, 2); // A->B
oGrafo.NovaAresta(0, 3, 1); // A->D
oGrafo.NovaAresta(1, 3, 3); // B->D
oGrafo.NovaAresta(1, 4, 10); // B->E
oGrafo.NovaAresta(2, 5, 5); // C->F
oGrafo.NovaAresta(2, 0, 4); // C->A
oGrafo.NovaAresta(3, 2, 2); // D->C
oGrafo.NovaAresta(3, 5, 8); // D->F
oGrafo.NovaAresta(3, 4, 2); // D->E
oGrafo.NovaAresta(3, 6, 4); // D->G
oGrafo.NovaAresta(4, 6, 6); // E->G
oGrafo.NovaAresta(6, 5, 1); // G->F
```



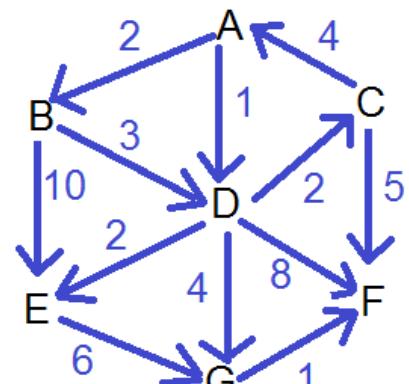
Quando o objeto da classe Grafo é criado, sua matriz de adjacências adjMatrix terá todos seus elementos com o valor infinity (um valor muito alto, maior que todas as distâncias possíveis na realidade modelada) e, assim, apenas as arestas existentes terão valores menores que infinity, após estas terem sido adicionadas através do método NovaAresta do objeto da classe Grafo.

Buscaremos um caminho entre os vértices dados pelos parâmetros inicioDoPercorso e finalDoPercorso.

As tabelas a seguir nos mostram o progresso feito pelo algoritmo enquanto ele trabalha através do grafo, partindo do vértice A (índice 0) e desejando chegar ao vértice G (índice 6).

A primeira tabela nos mostra os valores da tabela antes que o vértice A seja visitado (o valor Infinito indica que não sabemos a distância, e em código usaremos um valor tão grande que não representará um peso):

vertices[]		percurso[]	
rotulo	foiVisitado	distancia	verticePai
A	Falso	Infinito	0
B	Falso	Infinito	n/a
C	Falso	Infinito	n/a
D	Falso	Infinito	n/a
E	Falso	Infinito	n/a
F	Falso	Infinito	n/a
G	Falso	Infinito	n/a

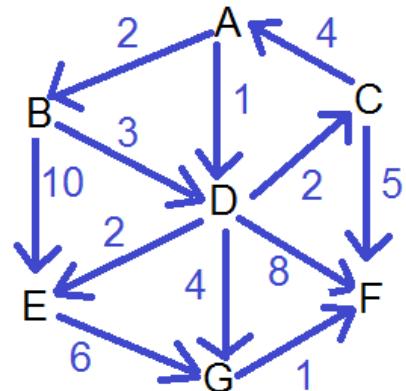


Após A ser visitado, a tabela será como abaixo. Note como, partindo de A, anotamos os vértices que possuem entrada direta a partir de A (B e D, pesos 2 e 1 respectivamente):

vertices[]		percurso[]	
rotulo	foiVisitado	distancia	verticePai
A	Verdadeiro	Infinito	0
B	Falso	2	A
C	Falso	Infinito	n/a
D	Falso	1	A
E	Falso	Infinito	n/a
F	Falso	Infinito	n/a
G	Falso	Infinito	n/a

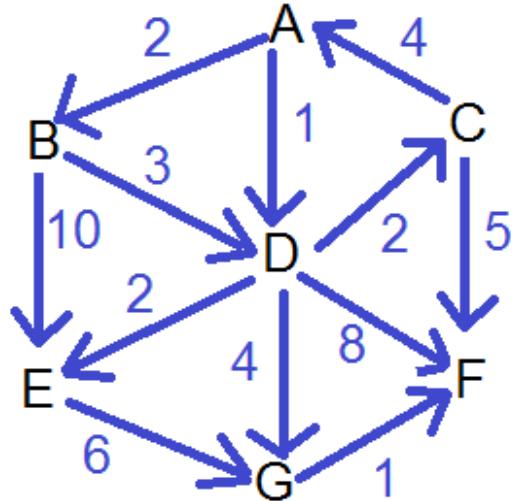
Após isso procuramos a saída de A com a menor distância, na tabela acima, que é a aresta para o vértice D, e o visitamos (passa a ser o atual e foiVisitado fica **true**). Para cada saída direta de D (C, E, F e G), somamos a distância dessa saída com o valor da distância de D vindo de A (ou seja, somamos 1 ao custo para cada um desses vértices a partir de D):

vertices[]		percurso[]	
rotulo	foiVisitado	distancia	verticePai
A	Verdadeiro	Infinito	0
B	Falso	2	A
C	Falso	2 + 1	D
D	Verdadeiro	1	A
E	Falso	2 + 1	D
F	Falso	8 + 1	D
G	Falso	4 + 1	D



Voltamos a buscar o vértice ainda não visitado com a menor distância desde A. Como o vértice D já foi visitado, o vértice B passa a ser aquele com a menor distância diretamente acessível a partir de A. No algoritmo, o método `ObterMenorDistancia()` devolverá o índice do vértice B, que visitaremos, portanto, em seguida:

vertices[]		percurso[]	
rotulo	foiVisitado	distancia	verticePai
A	Verdadeiro	Infinito	0
B	Verdadeiro	2	A
C	Falso	3	D
D	Verdadeiro	1	A
E	Falso	3	D
F	Falso	9	D
G	Falso	5	D



vertices[]		percurso[]	
rotulo	foiVisitado	distancia	verticePai
A	Verdadeiro	Infinito	0
B	Verdadeiro	2	A
C	Falso	3	D
D	Verdadeiro	1	A
E	Falso	3	D
F	Falso	8	C
G	Falso	5	D

Aqui, achou um caminho com menor distância e o troca, no método AjustarMenorCaminho()

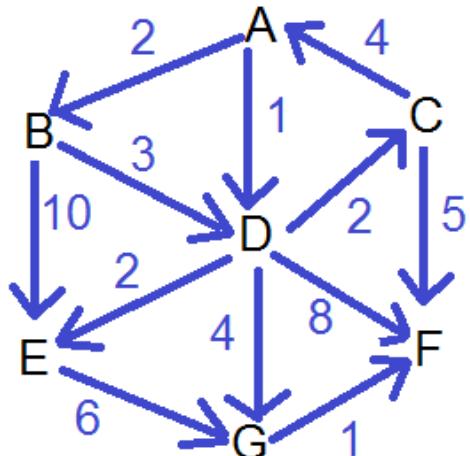
Agora, procuramos o vértice ainda não visitado com a menor distância no vetor percurso[]. Será o vértice E. Anotamos a distância desde o inicio (A) até E (3 unidades) e o marcamos como visitado, passando então a ajustar o menor caminho, caso encontremos distâncias ainda menores.

vertices[]		percurso[]	
rotulo	foiVisitado	distancia	verticePai
A	Verdadeiro	Infinito	0
B	Verdadeiro	2	A
C	Falso	3	D
D	Verdadeiro	1	A
E	Falso	3	D
F	Falso	6	G
G	Falso	5	D

Aqui, achou um caminho com menor distância e o troca, no método AjustarMenorCaminho()

O vértice ainda não visitado com a menor distância desde o início (A) será o vértice G. E assim prosseguiremos até que visitemos o

vertices[]		percurso[]	
rotulo	foiVisitado	distancia	verticePai
A	Verdadeiro	Infinito	0
B	Verdadeiro	2	A
C	Verdadeiro	3	D
D	Verdadeiro	1	A
E	Verdadeiro	3	D
F	Verdadeiro	6	G
G	Falso	5	D



último vértice, G:

Em seguida, exibiremos os percursos encontrados ou apenas aquele que nos interessa (de inicioDoPercorso até finalDoPercorso), usando o campo verticePai para percorrer, para trás, a tabela acima, desde finalDoPercorso até encontrarmos o vértice inicioDoPercorso.

Modificações no projeto para incorporar este algoritmo

Modifique o projeto que estamos fazendo, colocando um TabControl com duas abas. Ancore esse tabControl ao fundo e à direita, além das Âncoras já existentes. Na primeira aba, copie e cole os componentes visuais que já utilizamos anteriormente. Mude o nome dessa aba para tpFundamentos e seu Text para Fundamentos. Na segunda aba, mude seu nome para tpDijkstra e seu Text para Djikstra. A aparência da primeira guia do formulário ficará como abaixo e a **segunda guia** terá a aparência mostrada [nesta página](#) (2 labels, 2 textboxes, 1 botão e 1 listbox):

Z

Código para o Algoritmo de Dijkstra

O primeiro trecho de código modificado para o algoritmo é a classe Vertice, que estudamos anteriormente e que passará a ter o atributo lógico estaAtivo:

```
class Vertice
{
    public String rotulo;
    public bool foiVisitado;
    private bool estaAtivo;

    public Vertice(string nomeDoVertice)
    {
        rotulo = nomeDoVertice;
        foiVisitado = false;
        estaAtivo = true;
    }
}
```

Também precisaremos de uma classe que nos ajude a manter o rastreio do relacionamento entre um vértice distante e o vértice original usado para calcular os caminhos mais curtos. Esta classe é chamada de DistOriginal:

```
class DistOriginal
{
    public int distancia;
    public int verticePai;
    public DistOriginal(int vp, int d)
    {
        distancia = d;
        verticePai = vp;
    }
}
```

A classe Grafo terá as mudanças abaixo:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

class Grafo
{
    private const int NUM_VERTICES = 20;
    private Vertice[] vertices;
    private int[,] adjMatrix;
    int numVerts;
    DataGridView dgv; // para exibir a matriz de adjacência num formulário

    /// DIJKSTRA
    DistOriginal[] percurso;
    int infinity = int.MaxValue;
    int verticeAtual; // global que indica o vértice atualmente sendo visitado
    int doInicioAteAtual; // global usada para ajustar menor caminho com Djikstra
    int nTree;

    public Grafo(DataGridView dgv)
    {
        this.dgv = dgv;
```

```
vertices = new Vertice[NUM_VERTICES];
adjMatrix = new int[NUM_VERTICES, NUM_VERTICES];
numVerts = 0;
nTree = 0;

for (int j = 0; j < NUM_VERTICES; j++)      // zera toda a matriz
    for (int k = 0; k < NUM_VERTICES; k++)
        adjMatrix[j, k] = infinity; // distância tão grande que não existe

percurso = new DistOriginal[NUM_VERTICES];
}

public void NovoVertice(string label)
{
    vertices[numVerts] = new Vertice(label);
    numVerts++;
    if (dgv != null) // se foi passado um dataGridView para exibição
    {                // é realizado o seu ajuste para a quantidade de vértices
        dgv.RowCount = numVerts + 1;
        dgv.ColumnCount = numVerts + 1;
        dgv.Columns[numVerts].Width = 45;
    }
}

public void NovaAresta(int origem, int destino, int peso)
{
    adjMatrix[origem, destino] = peso; // sobrecarga do método anterior
}

// demais métodos anteriores : ordenação topológica, árvore geradora mínima,
// percursos - ajustar para comparar com infinity e não com zero
```

A classe Grafo que usamos anteriormente tem agora um novo conjunto de métodos para calcular os menores caminhos. O primeiro desses métodos é o Caminho(), que entabula os cálculos de caminhos mínimos:

```
public string Caminho(int inicioDoPercorso, int finalDoPercorso, ListBox lista)
{
    for (int j = 0; j < numVerts; j++)
        vertices[j].foiVisitado = false;

    vertices[inicioDoPercorso].foiVisitado = true;
    for (int j = 0; j < numVerts; j++)
    {
        // anotamos no vetor percurso a distância entre o inicioDoPercorso e cada vértice
        // se não há ligação direta, o valor da distância será infinity
        int tempDist = adjMatrix[inicioDoPercorso, j];
        percurso[j] = new DistOriginal(inicioDoPercorso, tempDist);
    }

    for (int nTree = 0; nTree < numVerts; nTree++)
    {
        // Procuramos a saída não visitada do vértice inicioDoPercorso com a menor distância
        int indiceDoMenor = ObterMenor();

        // e anotamos essa menor distância
        int distanciaMinima = percurso[indiceDoMenor].distancia;
```

```
// o vértice com a menor distância passa a ser o vértice atual
// para compararmos com a distância calculada em AjustarMenorCaminho()
verticeAtual = indiceDoMenor;
doInicioAteAtual = percurso[indiceDoMenor].distancia;

// visitamos o vértice com a menor distância desde o inicioDoPercorso
vertices[verticeAtual].foiVisitado = true;
AjustarMenorCaminho(lista);
}

return ExibirPercursos(inicioDoPercorso, finalDoPercorso, lista);
}
```

Este método usa outros métodos auxiliares, **ObterMenor** e **AjustarMenorCaminho**. O loop **for** no começo do método busca os vértices alcançáveis a partir do vértice inicial e os coloca no vetor **percurso**. Esse vetor armazena as distâncias mínimas a partir dos diferentes vértices e eventualmente armazenará os menores caminhos finais.

O loop principal (o segundo for) realiza três operações:

1. Encontra a entrada no vetor **percurso** com a menor distância
2. Torna esse vértice o vértice atual
3. Atualiza o vetor **percurso** para mostrar as distâncias a partir do vértice atual.

A maioria desse trabalho é feito pelos métodos **obterMenor** e **ajustarMenorCaminho**:

```
public int ObterMenor()
{
    int distanciaMinima = infinity;
    int indiceDaMinima = 0;
    for (int j = 0; j < numVerts; j++)
        if (!(vertices[j].foiVisitado) && (percurso[j].distancia < distanciaMinima))
    {
        distanciaMinima = percurso[j].distancia;
        indiceDaMinima = j;
    }
    return indiceDaMinima;
}

public void AjustarMenorCaminho(ListBox lista)
{
    for (int coluna = 0; coluna < numVerts; coluna++)
        if (!vertices[coluna].foiVisitado)          // para cada vértice ainda não visitado
    {
        // acessamos a distância desde o vértice atual (pode ser infinity)
        int atualAteMargem = adjMatrix[verticeAtual, coluna];

        // calculamos a distância desde inicioDoPercorso passando por vertice atual
        // até esta saída
        int doInicioAteMargem = doInicioAteAtual + atualAteMargem;

        // quando encontra uma distância menor, marca o vértice a partir do
        // qual chegamos no vértice de índice coluna, e a soma da distância
        // percorrida para nele chegar
        int distanciaDoCaminho = percurso[coluna].distancia;
        if (doInicioAteMargem < distanciaDoCaminho)
        {
            percurso[coluna].verticePai = verticeAtual;
            percurso[coluna].distancia = doInicioAteMargem;
            ExibirTabela(lista);
        }
    }
}
```

```
        }
    }
    lista.Items.Add("=====Caminho ajustado=====");
    lista.Items.Add(" ");
}
public void ExibirTabela(ListBox lista)
{
    string dist = "";
    lista.Items.Add("Vértice\tVisitado?\tPeso\tVindo de");
    for (int i = 0; i < numverts; i++)
    {
        if (percurso[i].distancia == infinity)
            dist = "inf";
        else
            dist = Convert.ToString(percurso[i].distancia);

        lista.Items.Add(vertices[i].rotulo + "\t" + vertices[i].foiVisitado +
                        "\t\t" + dist + "\t" + vertices[percurso[i].verticePai].rotulo);
    }
    lista.Items.Add("-----");
}
```

O método **ObterMenor** percorre o vetor **percurso** até que a distância mínima é determinada, a qual é, então, retornada pelo método. O método **AjustarMenorCaminho** toma um novo vértice, encontra o próximo conjunto de vértices conectados a este vértice, calcula os menores caminhos e atualiza o vetor **percurso** até que uma distância menor é encontrada.

```
public string ExibirPercursos(int inicioDoPercuso, int finalDoPercuso,
                               ListBox lista)
{
    string resultado = "";
    for (int j = 0; j < numverts; j++)
    {
        resultado += vertices[j].rotulo + "=";
        if (percurso[j].distancia == infinity)
            resultado += "inf";
        else
            resultado += percurso[j].distancia + " ";
        string pai = vertices[percurso[j].verticePai].rotulo;
        resultado += "(" + pai + " ) ";
    }
    lista.Items.Add(resultado);
    lista.Items.Add(" ");
    lista.Items.Add(" ");
    lista.Items.Add("Caminho entre " + vertices[inicioDoPercuso].rotulo +
                  " e " + vertices[finalDoPercuso].rotulo);
    lista.Items.Add(" ");

    int onde = finalDoPercuso;
    Stack<string> pilha = new Stack<string>();

    int cont = 0;
    while (onde != inicioDoPercuso)
    {
        onde = percurso[onde].verticePai;
        pilha.Push(vertices[onde].rotulo);
        cont++;
    }
}
```

```
resultado = "";
while (pilha.Count != 0)
{
    resultado += pilha.Pop();
    if (pilha.Count != 0)
        resultado += " --> ";
}

if ((cont == 1) && (percurso[finalDoPercorso].distancia == infinity))
    resultado = "Não há caminho";
else
    resultado += " --> " + vertices[finalDoPercorso].rotulo;
return resultado;
}
```

Finalmente, o método **ExibirPercursos** acima mostra o conteúdo final do vetor **percurso**.

Para tornar o grafo disponível para outros algoritmos da classe Grafo, a variável **nTree** é configurada como 0 e as flags **foiVisitado** são todas configuradas para false.

Para colocar tudo isso em contexto, segue o método principal de uma aplicação console que usa o código para calcular os menores caminhos usando o algoritmo de Dijkstra, para testar essa implementação:

```
class capituloGrafos
{
    static void Main()
    {
        Grafo oGrafo = new Grafo(null);
        oGrafo.NovoVertice("A");           // 0
        oGrafo.NovoVertice("B");           // 1
        oGrafo.NovoVertice("C");           // 2
        oGrafo.NovoVertice("D");           // 3
        oGrafo.NovoVertice("E");           // 4
        oGrafo.NovoVertice("F");           // 5
        oGrafo.NovoVertice("G");           // 6
        oGrafo.NovaAresta(0, 1, 2);       // A --> B = 2
        oGrafo.NovaAresta(0, 3, 1);       // A --> D = 1
        oGrafo.NovaAresta(1, 3, 3);       // B --> D = 3
        oGrafo.NovaAresta(1, 4, 10);      // B --> E = 10
        oGrafo.NovaAresta(2, 5, 5);       // C --> F = 5
        oGrafo.NovaAresta(2, 0, 4);       // C --> A = 4
        oGrafo.NovaAresta(3, 2, 2);       // D --> C = 2
        oGrafo.NovaAresta(3, 5, 8);       // D --> F = 8
        oGrafo.NovaAresta(3, 4, 2);       // D --> E = 2
        oGrafo.NovaAresta(3, 6, 4);       // D --> G = 4
        oGrafo.NovaAresta(4, 6, 6);       // E --> G = 6
        oGrafo.NovaAresta(6, 5, 1);       // G --> F = 1
        lsbCaminho.Items.Clear();
        lsbCaminho.Items.Add(" ");
        lsbCaminho.Items.Add("Menores caminhos:");
        lsbCaminho.Items.Add(" ");
        int inicio = int.Parse(txtInicio.Text);
        int fim = int.Parse(txtFim.Text);
        lsbCaminho.Items.Add(oGrafo.Caminho(inicio, fim, lsbCaminho));
        lsbCaminho.Items.Add(" ");
    }
}
```

O resultado desse programa, indo do vértice A para o G, é:

Operações com Grafos

Fundamentos Dijkstra

Origem Destino Buscar

Menores caminhos:

Vértice	Visitado?	Peso	Vindo de
A	True	inf	A
B	False	2	A
C	False	3	D
D	True	1	A
E	False	inf	A
F	False	inf	A
G	False	inf	A

Vértice	Visitado?	Peso	Vindo de
A	True	inf	A
B	False	2	A
C	False	3	D
D	True	1	A
E	False	3	D
F	False	inf	A
G	False	inf	A

Vértice	Visitado?	Peso	Vindo de
A	True	inf	A
B	False	2	A
C	False	3	D
D	True	1	A
E	False	3	D
F	False	9	D
G	False	inf	A

Vértice	Visitado?	Peso	Vindo de
A	True	inf	A
B	False	2	A
C	False	3	D
D	True	1	A
E	False	3	D
F	False	9	D
G	False	5	D

=====Caminho ajustado=====

=====Caminho ajustado=====

Vértice Visitado? Peso Vindo de

A True inf A

B True 2 A

C True 3 D

D True 1 A

E True 3 D

F False 6 G

G True 5 D

=====Caminho ajustado=====

=====Caminho ajustado=====

=====Caminho ajustado=====

A=inf(A) B=2(A) C=3(D) D=1(A) E=3(D) F=6(G) G=5(D)

Caminho entre A e G

A--> D --> G

Operações com Grafos

Fundamentos Dijkstra

Origem Destino Buscar

Vértice	Visitado?	Peso	Vindo de
D	True	1	A
E	False	3	D
F	False	8	C
G	False	5	D

=====Caminho ajustado=====

=====Caminho ajustado=====

Vértice	Visitado?	Peso	Vindo de
A	True	inf	A
B	True	2	A
C	True	3	D
D	True	1	A
E	True	3	D
F	False	6	G
G	True	5	D

=====Caminho ajustado=====

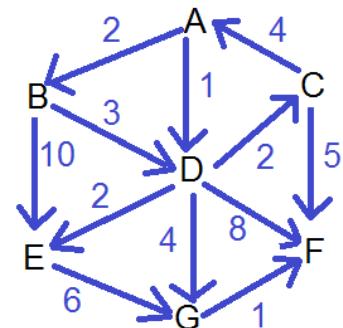
=====Caminho ajustado=====

=====Caminho ajustado=====

A=inf(A) B=2(A) C=3(D) D=1(A) E=3(D) F=6(G) G=5(D)

Caminho entre A e G

A--> D --> G



Partindo do vértice C para o E, temos:

Operações com Grafos

Fundamentos Djikstra

Origem **2** Destino **4** Buscar

Menores caminhos:

Vértice	Visitado?	Peso	Vindo de
A	True	4	C
B	False	6	A
C	True	inf	C
D	False	inf	C
E	False	inf	C
F	False	5	C
G	False	inf	C

Vértice	Visitado?	Peso	Vindo de
A	True	4	C
B	False	6	A
C	True	inf	C
D	False	5	A
E	False	inf	C
F	False	5	C
G	False	inf	C

=====Caminho ajustado=====

Vértice	Visitado?	Peso	Vindo de
A	True	4	C
B	False	6	A
C	True	inf	C
D	True	5	A
E	False	7	D
F	False	5	C
G	False	inf	C

Vértice	Visitado?	Peso	Vindo de
A	True	4	C
B	False	6	A
C	True	inf	C
D	True	5	A
E	False	7	D
F	False	5	C
G	False	9	D

=====Caminho ajustado=====

=====Caminho ajustado=====

=====Caminho ajustado=====

=====Caminho ajustado=====

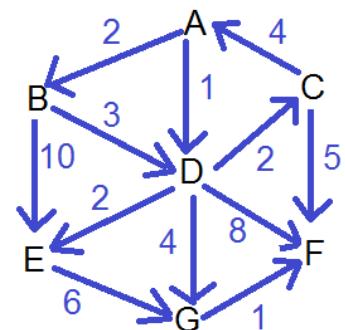
=====Caminho ajustado=====

=====Caminho ajustado=====

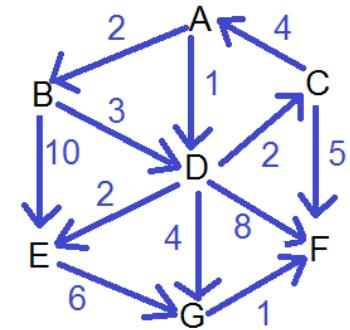
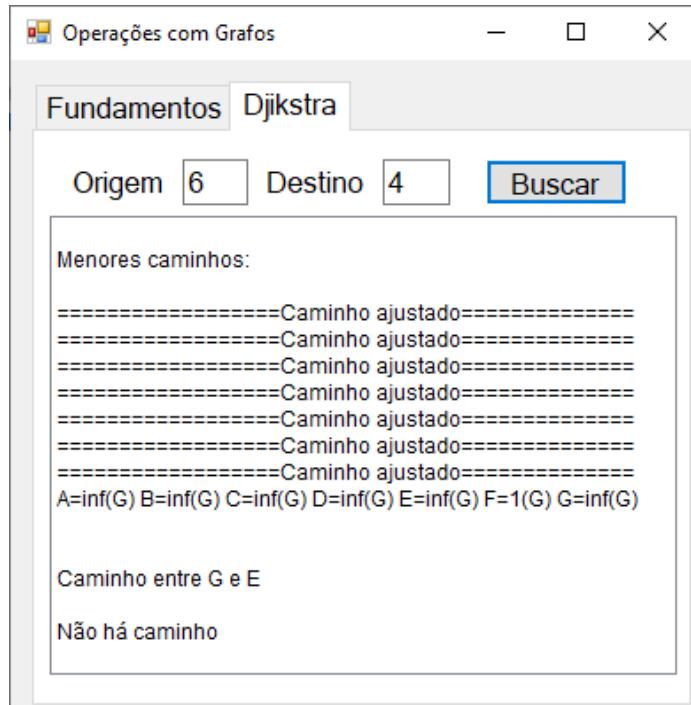
A=4(C) B=6(A) C=inf(C) D=5(A) E=7(D) F=5(C) G=9(D)

Caminho entre C e E

C → A → D → E



Finalmente, partindo de G, que tem apenas uma saída para um vértice sem nenhuma saída (F), e tentando chegar em E, temos a figura a seguir.



Estude o algoritmo de exibição e faça com que não seja exibida uma saída quando não houver caminho.

Resumo

Grafos são uma das mais importantes estruturas de dados usadas em Ciência da Computação.

São usados regularmente para modelar todas as coisas, desde circuitos elétricos até cronogramas de cursos em universidades e rotas de caminhões e linhas aéreas.

Weiss (1999) contém uma discussão mais técnica dos algoritmos de grafos abordados neste capítulo, enquanto LaFore (1998) contém explicações práticas muito boas de todos os algoritmos abordados aqui.

EXERCÍCIOS

1. Construa uma grafo ponderado que modele uma seção do estado onde você mora. Use o algoritmo de Dijkstra's para determinar o menor caminho de um vértice inicial até um vértice final.
2. A partir dos pesos do grafo do Exercício 1 construa a árvore geradora mínima.
3. Ainda usando o grafo do Exercício 1, escreva uma aplicação Windows que permita ao usuário buscar por um vértice no grafo usando tanto o percurso em profundidade quanto o percurso em largura.

10. Ordenação

As duas operações mais comuns feitas sobre dados armazenados em computadores são a ordenação e a busca. Isso tem sido verdadeiro desde o início da indústria de computação, o que significa que ordenação e busca são duas das operações mais estudadas em Ciência da Computação. Muitas das estruturas de dados que estudamos são projetadas primariamente para tornar a ordenação e/ou a pesquisa mais fáceis e mais eficientes sobre os dados que a estrutura armazena.

Neste capítulo estudaremos os algoritmos fundamentais de ordenação e busca de dados. Estes algoritmos usam apenas os vetores como estruturas de dados e a única técnica de programação "avanhada" que usaremos é a recursão.

Este capítulo também apresenta as técnicas que usaremos para analisar, informalmente, diferentes algoritmos em termos de velocidade e eficiência.

A maioria dos dados que usamos no dia-a-dia vem ordenados. Procuramos palavras e suas definições no dicionário através de uma busca alfabética. Buscamos um número de telefone percorrendo os sobrenomes na lista telefônica também alfabeticamente. O correio ordena as correspondências em diversas maneiras - pelo CEP, em seguida pelo endereço e finalmente pelo nome. Ordenação é um processo fundamental quando trabalhamos com dados e merece um estudo mais aprofundado.

Como dito anteriormente, tem havido uma grande quantidade de pesquisa sobre diferentes técnicas de ordenação. Embora alguns algoritmos de ordenação bastante sofisticados tenham sido desenvolvidos, há também vários algoritmos simples de ordenação que você deveria estudar antes. Esses algoritmos são a ordenação por inserção (insertion sort), a ordenação por bolha (bubble sort) e a ordenação por seleção direta (selection sort). Cada um desses algoritmos é fácil de entender e fácil de implementar. Eles não são, de qualquer maneira, os melhores algoritmos para ordenação, mas para pequenos conjuntos de dados e em outras circunstâncias especiais, eles são os melhores algoritmos para usar.

1. Uma classe de Testes para Vetores

Para examinar esses algoritmos, primeiramente precisamos de uma "bancada de testes" na qual implementaremos e testaremos os algoritmos.

Primeiramente, vamos criar um novo projeto no Visual Studio, com o nome OrdenacoesSimples. Em seguida, dentro desse projeto criaremos uma classe que encapsula as operações normais realizadas com um vetor — inserção de elemento, acesso de elemento e exibição do conteúdo do vetor. Eis o código da classe:

```
class CArray
{
    private int[] arr;
    private int upper;
    private int numElements;
    DataGridView dgv;
    bool animar;

    public CArray(int size, DataGridView dgvDoFormulario, bool animacao)
    {
        arr = new int[size];
        upper = size - 1;
        numElements = 0;
        animar = animacao;

        dgv = dgvDoFormulario;
```

```
if (animar)
{
    dgv.ColumnCount = size;
    dgv.RowCount = 1;

    for (int indice = 0; indice < size; indice++)
    {
        dgv.Columns[indice].Name = Convert.ToString(indice);
        dgv.Columns[indice].Width = 25;
    }
}

public void Insert(int item)
{
    arr[numElements] = item;
    if (animar)
        dgv.Rows[0].Cells[numElements].Value = Convert.ToString(item);
    numElements++;
}

public void DisplayElements()
{
    for (int i = 0; i <= upper; i++)
        Console.WriteLine(arr[i] + " ");
}

public void Clear()
{
    for (int i = 0; i <= upper; i++)
        arr[i] = 0;

    dgv.ColumnCount = 0;
}

public void BubbleSort()
{
    if (animar)
    {
        dgv.Parent.Height = 165 + 22;
        dgv.Rows.Add();           // usado para organizar o DataGridView para exibição
    }
    int temp, linha = 0;      // linha será usada para exibir as trocas de valores
    for (int outer = upper; outer >= 0; outer--)
    {
        for (int inner = 0; inner <= outer - 1; inner++)
            if (arr[inner] > arr[inner + 1])
            {
                if (animar)
                {
                    dgv.Parent.Height += 22;
                    linha = dgv.Rows.Add();
                    for (int indice = 0; indice < numElements; indice++)
                        dgv.Rows[linha-1].Cells[indice].Value = Convert.ToString(arr[indice]);

                    dgv.Rows[linha - 1].Cells[inner].Style.BackColor = Color.Red;
                    dgv.Rows[linha - 1].Cells[inner + 1].Style.BackColor = Color.Orange;
                    Application.DoEvents();
                }
                temp = arr[inner];
                arr[inner] = arr[inner + 1];
                arr[inner + 1] = temp;
            }
    }
}
```

```
        }
        if (animar)
        {
            for (int indice = 0; indice < numElements; indice++)
                dgv.Rows[0].Cells[indice].Value = dgv.Rows[linha + 1].Cells[indice].Value;
            dgv.RowCount--;
        }
    }

    public void SelectionSort()
{
    if (animar)
    {
        dgv.Parent.Height = 165 + 22;
        dgv.Rows.Add();           // usado para organizar o DataGridView para exibição
    }
    int min, temp, linha = 0;   // linha será usada para exibir as trocas de valores
    for (int outer = 0; outer <= upper; outer++)
    {
        min = outer;
        if (animar)
        {
            dgv.Parent.Height += 22;
            linha = dgv.Rows.Add();
            for (int indice = 0; indice < numElements; indice++)
                dgv.Rows[linha - 1].Cells[indice].Value = Convert.ToString(arr[indice]);
        }
        for (int inner = outer + 1; inner <= upper; inner++)
            if (arr[inner] < arr[min])
                min = inner;
        temp = arr[outer];
        arr[outer] = arr[min];
        arr[min] = temp;

        if (animar)
        {
            dgv.Rows[linha].Cells[min].Style.BackColor = Color.Red;
            dgv.Rows[linha].Cells[outer].Style.BackColor = Color.Orange;
            Application.DoEvents();
        }
    }
}

public void InsertionSort()
{
    if (animar)
    {
        dgv.Parent.Height = 165 + 22;
        dgv.Rows.Add();           // usado para organizar o DataGridView para exibição
    }

    int inner, temp, linha = 0;
    for (int outer = 1; outer < numElements; outer++)
    {
        temp = arr[outer];
        inner = outer;
        if (animar)
        {
            dgv.Parent.Height += 22;
            linha = dgv.Rows.Add();
            for (int indice = 0; indice < numElements; indice++)
                dgv.Rows[linha - 1].Cells[indice].Value = Convert.ToString(arr[indice]);
        }
    }
}
```

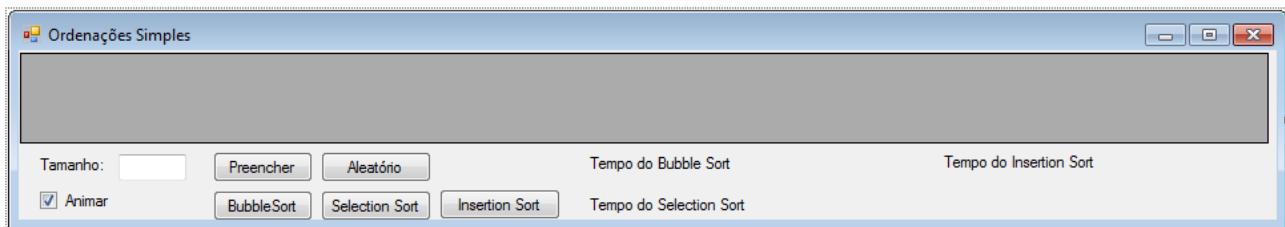
```
        while (inner > 0 && arr[inner - 1] > temp)
    {
        arr[inner] = arr[inner - 1];
        inner -= 1;
    }
    arr[inner] = temp;

    if (animar)
    {
        dgv.Rows[linha - 1].Cells[inner].Style.BackColor = Color.Red;
        dgv.Rows[linha - 1].Cells[outer].Style.BackColor = Color.Orange;
        Application.DoEvents();
    }
}

if (animar)
    for (int indice = 0; indice < numElements; indice++)
        dgv.Rows[linha].Cells[indice].Value = Convert.ToString(arr[indice]);
}

public CArray Copiar()
{
    CArray copia = new CArray(numElements, dgv, animar);
    for (int indice = 0; indice < numElements; indice++)
        copia.Insert(arr[indice]);
    return copia;
}
}
```

No formulário, coloque um DataGridView (coleção Data), chame-o de dgvVetor e ancore-o com a lateral direita do formulário. Coloque um label, com Text "Tamanho:" e um textBox, com nome txtTamanho. Coloque também um botão, com nome btnPreencher, sob o DataGridView e mude a propriedade Text do Formulário para "Ordenações Simples". O nome do formulário deve ser mudado para frmOrdenacoesSimples e sua aparência será semelhante à da figura abaixo:



No evento OnClick do btnPreencher, digite o seguinte:

```
CArray nums; // variável global
...
private void btnPreencher_Click(object sender, EventArgs e)
{
    int tamanho = Convert.ToInt32(txtTamanho.Text);
    nums = new CArray(tamanho, dgvVetor, chkAnimar.Checked);
    for (int i = 0; i < tamanho; i++)
        nums.Insert(i);
}
```

A execução, com o pressionamento do botão, será semelhante a :



Antes de deixarmos a classe CArray para iniciarmos o exame dos algoritmos de ordenação e de pesquisa, vamos discutir como estaremos realmente armazenando os dados num objeto da classe CArray. A fim de demonstrar mais efetivamente como os diferentes algoritmos de ordenação se comportam, os dados no vetor precisam estar numa ordem aleatória. Isto será obtido pelo uso de um gerador de números aleatórios para atribuir a cada um dos elementos do vetor.

Números aleatórios podem ser criados em C# com a classe Random. Um objeto deste tipo pode gerar números aleatórios. Para instanciar um objeto da classe Random , você deve passar uma semente para o construtor da classe. Esta semente pode ser vista como como um limite superior para o intervalo de números que o gerador de números aleatórios poderá criar.

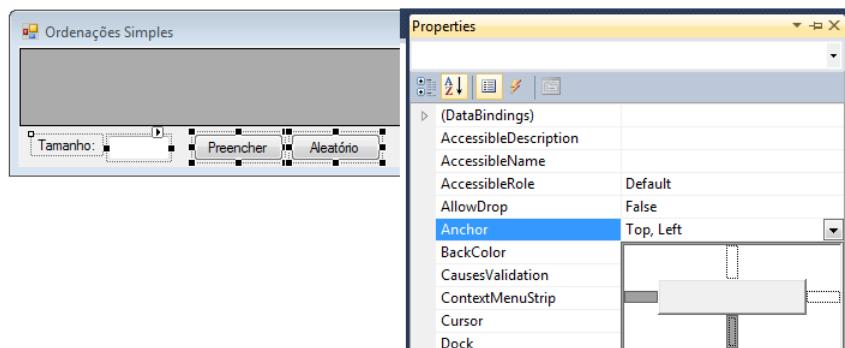
Adicionaremos um novo botão no formulário, chamando-o de btnAleatorio e, no seu evento OnClick usaremos números aleatórios para povoar os elementos do vetor:

```
private void btnAleatorio_Click(object sender, EventArgs e)
{
    int tamanho = Convert.ToInt32(txtTamanho.Text);
    nums = new CArray(tamanho, dgvVetor, chkAnimar.Checked);
    Random rnd = new Random(1000);
    for (int i = 0; i < tamanho; i++)
        nums.Insert(rnd.Next(tamanho));
}
```

A saída do programa agora, com 34 elementos, por exemplo, seria:



Libere as âncoras superiores e fixe as âncoras inferiores de todos os componentes (menos o DataGridView) de forma que, se o formulário tiver sua altura aumentada, esses componentes acompanhem a parte inferior do formulário. Fixe também a âncora inferior do DataGridView.



2. Bubble Sort

O primeiro algoritmo de ordenação que examinaremos é o bubble sort. O bubble sort é um dos mais lentos algoritmos de ordenação disponíveis, mas é também um dos mais simples de entender e implementar, o que o torna um excelente candidato para nosso primeiro algoritmo de ordenação.

A ordenação por bolha obtém seu nome porque os valores “flutuam como uma bolha” de uma ponta da lista para a outra ponta. Assumindo que você está ordenando uma lista de números em ordem crescente, valores maiores flutuam para a direita enquanto valores menores flutuam para a esquerda (com se fosse um vidro de xampu cheio de bolhas de ar, onde as maiores se dirigem para cima e as menores para baixo).

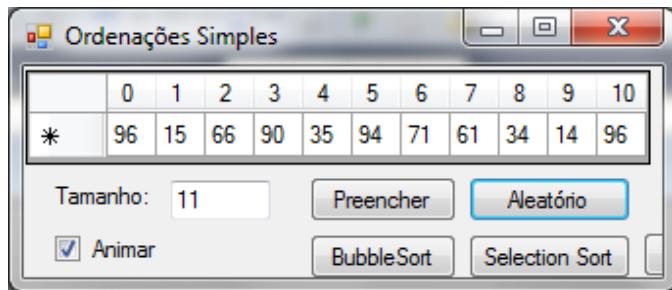
Este comportamento é causado pela movimentação através da lista várias vezes, comparando valores adjacentes e trocando-os se o valor à esquerda é maior do que o valor à direita.

A figura [abaixo](#) ilustra como o bubble sort funciona. Dois números do vetor (2 e 72) estão enfatizados com círculos. Observe como 72 se move do inicio para o meio do vetor, enquanto 2 se move de logo depois do meio para o início do vetor:

(72)	54	59	30	31	78	(2)	77	82	72
54	58	30	31	(72)	(2)	77	78	72	82
54	30	32	58	(2)	(72)	72	77	78	82
30	32	54	(2)	58	(72)	72	77	78	82
30	32	(2)	54	58	(72)	72	77	78	82
30	(2)	32	54	58	(72)	72	77	78	82
(2)	30	32	54	58	(72)	72	77	78	82

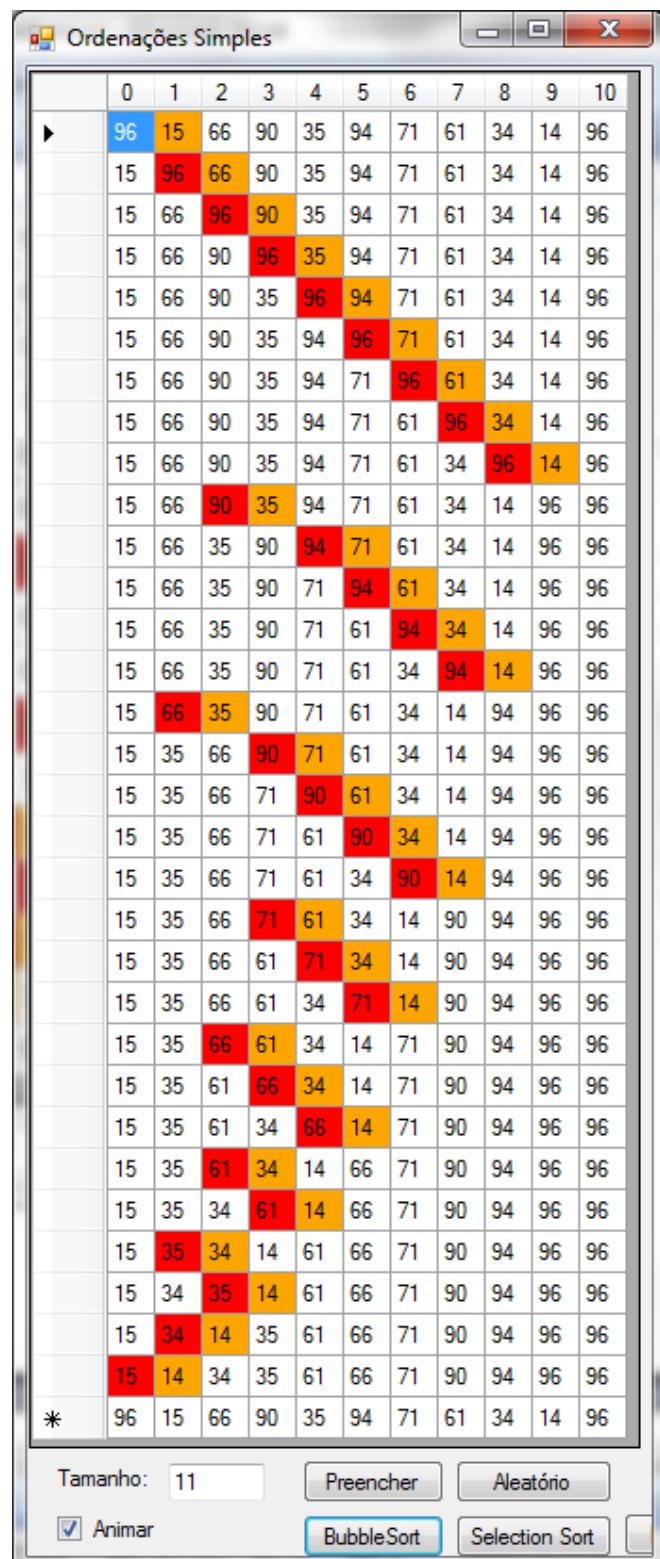
O código para o algoritmo BubbleSort vem a seguir:

```
public void BubbleSort()
{
    int temp, linha = 0; // linha será usada para exibir as trocas de valores
    for (int outer = upper; outer >= 1; outer--)
    {
        for (int inner = 0; inner <= outer - 1; inner++)
            if (arr[inner] > arr[inner + 1])
            {
                temp = arr[inner];
                arr[inner] = arr[inner + 1];
                arr[inner + 1] = temp;
            }
    }
}
```



Vamos alterar o método acima, de forma que cada troca crie uma nova linha no DataGridView e as posições envolvidas nas trocas sejam identificadas por cores vermelha e laranja.

Os comandos abaixo devem ser incluídos antes dos comandos de troca, logo após o if que determina a necessidade de trocar elementos. O código virá na próxima página e



Há muitas coisas para observar no algoritmo de ordenação por bolha. Em primeiro lugar, o código para trocar dois elementos é escrito dentro do método, ao invés de ser um método separado. Um método separado para trocas tornará mais lenta a ordenação, pois ele será chamado várias vezes e, a cada chamada, haverá gasto de tempo com passagem de

parâmetros e tratamento do endereço de retorno.

Já que o código de troca tem somente três linhas de comprimento, a clareza do código não foi sacrificada pela não criação de uma rotina separada para trocas.

Ainda mais importante, note que a repetição externa se inicia ao final do vetor e o seu índice se move em direção ao início do vetor. Se você olhar novamente para a figura no início desta página, verá que o maior valor do vetor está em seu lugar adequado no final do vetor.

Isto significa que os índices do vetor que são maiores do que os valores da repetição externa já estão em seus lugares adequados e o algoritmo não precisa mais acessar esses valores.

A repetição interna começa no primeiro elemento do vetor e termina quando encontra o índice outer. A repetição interna compara duas posições adjacentes, indicadas por inner e inner +1, trocando-as se necessário.

Na última linha do DataGridView, temos o vetor original para compararmos com a penúltima linha, onde ele se encontra ordenado.

```
public void BubbleSort()
{
    if (animar)
    {
        dgv.Parent.Height = 165 + 22;
        dgv.Rows.Add();           // usado para organizar o DataGridView para exibição
    }

    int temp, linha = 0;          // linha será usada para exibir as trocas de valores
    for (int outer = upper; outer >= 0; outer--)
    {
        for (int inner = 0; inner <= outer - 1; inner++)
            if (arr[inner] > arr[inner + 1])
            {
                if (animar)
                {
                    dgv.Parent.Height += 22;
                    linha = dgv.Rows.Add();
                    for (int indice = 0; indice < numElements; indice++)
                        dgv.Rows[linha - 1].Cells[indice].Value = Convert.ToString(arr[indice]);

                    dgv.Rows[linha - 1].Cells[inner].Style.BackColor = Color.Red;
                    dgv.Rows[linha - 1].Cells[inner + 1].Style.BackColor = Color.Orange;
                    Application.DoEvents();
                }

                temp = arr[inner];
                arr[inner] = arr[inner + 1];
                arr[inner + 1] = temp;
            }
    }

    if (animar)
    {
        dgv.Parent.Height += 22;
        linha = dgv.Rows.Add();
        for (int indice = 0; indice < numElements; indice++)
            dgv.Rows[linha - 1].Cells[indice].Value = Convert.ToString(arr[indice]);

        for (int indice = 0; indice < numElements; indice++)
            dgv.Rows[0].Cells[indice].Value = dgv.Rows[linha + 1].Cells[indice].Value;
        dgv.RowCount--;
    }
}
```

No formulário, o evento OnClick do btnBubbleSort será:

```
private void btnBubbleSort_Click(object sender, EventArgs e)
{
    nums.BubbleSort();
}
```

3. Selection Sort

A próxima ordenação a examinarmos é a Ordenação por Seleção Direta, ou Straight-Select Sort. Esta ordenação começa com um índice no início do vetor, comparando o primeiro elemento com os outros elementos do vetor. O menor elemento é colocado na posição 0, e então a ordenação recomeça na posição 1. Isto continua até que cada posição, exceto a última, tenha sido o início de um novo loop de comparações.

Duas repetições são usadas no algoritmo SelectionSort. O loop externo move do primeiro elemento do vetor até o penúltimo elemento, enquanto o loop interno se move do segundo elemento do vetor até o último elemento, procurando por valores que sejam menores que o elemento atualmente armazenado na posição indexada pelo loop externo. Depois de cada iteração do loop interno, o menor de todos os valores do vetor é atribuído para seu local adequado. A figura abaixo ilustra como esse algoritmo funcionaria com os dados usados anteriormente na explicação sobre BubbleSort.

72	54	59	30	31	78	2	77	82	72
2	54	59	30	31	78	72	77	82	72
2	30	59	54	31	78	72	77	82	72
2	30	31	54	59	78	72	77	82	72
2	30	31	54	59	78	72	77	82	72
2	30	31	54	59	78	72	77	82	72
2	30	31	54	59	72	78	77	82	72
2	30	31	54	59	72	72	77	82	78
2	30	31	54	59	72	72	77	82	78
2	30	31	54	59	72	72	77	78	82

O código que implementa o algoritmo SelectionSort é exibido abaixo:

```
public void SelectionSort()
{
    int min, temp;
    for (int outer = 0; outer <= upper; outer++)
    {
        min = outer;
        for (int inner = outer + 1; inner <= upper; inner++)
            if (arr[inner] < arr[min])
                min = inner;
        temp = arr[outer];
        arr[outer] = arr[min];
        arr[min] = temp;
    }
}
```

Esse é, basicamente, o algoritmo que aprendemos no 1º semestre do curso, na disciplina Técnicas de Programação. Ao invés de outer e inner, usávamos como nomes de índices as variáveis Lento e Rapido, respectivamente. Para permitir a apresentação do vetor a cada troca de valores, faremos modificações de acordo com o código abaixo:

```
public void SelectionSort()
{
    if (animar)
    {
        dgv.Parent.Height = 165 + 22;
        dgv.Rows.Add(); // usado para organizar o DataGridView para exibição
    }

    int min, temp, linha = 0; // linha será usada para exibir as trocas de valores
    for (int outer = 0; outer <= upper; outer++)
    {
        min = outer;
        if (animar)
        {
            dgv.Parent.Height += 22;
            linha = dgv.Rows.Add();
            for (int indice = 0; indice < numElements; indice++)
                dgv.Rows[linha - 1].Cells[indice].Value = Convert.ToString(arr[indice]);
        }
        for (int inner = outer + 1; inner <= upper; inner++)
            if (arr[inner] < arr[min])
                min = inner;

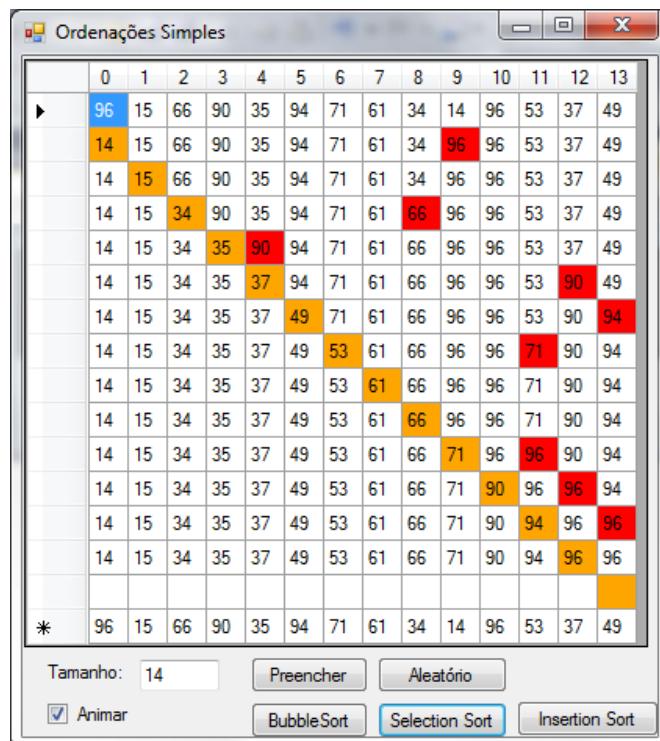
        temp = arr[outer];
        arr[outer] = arr[min];
        arr[min] = temp;

        if (animar)
        {
            dgv.Rows[linha].Cells[min].Style.BackColor = Color.Red;
            dgv.Rows[linha].Cells[outer].Style.BackColor = Color.Orange;
            Application.DoEvents();
        }
    }
}
```

O resultado da execução para 14 valores está ao lado. Observe que o número de trocas foi bem menor e que a execução do algoritmo foi bem mais rápida que o BubbleSort.

Um pergunta: por que os números 15 e 61 não aparecem em vermelho em nenhum momento?

O algoritmo básico final que veremos neste capítulo é o mais simples de entender: a ordenação por Inserção, ou Insertion Sort.



4. Insertion Sort

A ordenação por Inserção é semelhante à maneira que nós normalmente usamos para ordenar coisas numérica ou alfabeticamente. Digamos que eu tenha que ordenar alfabeticamente uma relação de carteiras estudantis com RAs, nomes e classe de alunos. A secretaria me entregou as carteiras em uma ordem aleatória (claro, que se podia esperar?), mas eu gostaria que elas fossem organizadas em ordem alfabética pelo nome do aluno, para que eu já pudesse fazer a chamada enquanto as entrego, além de indicar, pela ordem de chamada, em que computador cada aluno deverá se sentar daqui para frente.

Eu levo as carteiras para minha sala (se eu tivesse uma sala), limpo minha mesa (se eu tivesse uma mesa) e pego a primeira carteira.

O nome nessa carteira é Sabrina. Eu a coloco no canto superior esquerdo da mesa e pego a segunda carteira, cujo nome é Caique. Eu movo a carteira de Sabrina para a direita e coloco a carteira de Caique no lugar onde estava a carteira de Sabrina. A próxima carteira é do Wesley. Ele pode ser inserido à direita sem que eu tenha de deslocar nenhuma outra carteira. A próxima carteira é de Ana Laura.

Ela tem que ir para o começo da lista, portanto cada uma das outras carteiras devem ser deslocadas uma posição para a direita, a fim de gerar espaço. Esta é a maneira pela qual o Insertion sort funciona.

O código para o Insertion Sort é mostrado abaixo, seguindo uma explicação sobre seu funcionamento:

```
public void InsertionSort()
{
    int inner, temp;
    for (int outer = 1; outer <= upper; outer++)
    {
        temp = arr[outer];
        inner = outer;
```

```
        while (inner > 0 && arr[inner - 1] >= temp)
    {
        arr[inner] = arr[inner - 1];
        inner -= 1;
    }
    arr[inner] = temp;
}
}
```

O Insertion sort tem dois loops. O loop externo percorre elemento a elemento através do vetor, enquanto o loop interno compara o elemento escolhido no loop externo com o elemento seguinte a ele no vetor. Se o elemento selecionado pelo loop externo é menor que o elemento selecionado pelo loop interno, elementos do vetor são deslocados para a direita com o intuito de prover espaço para o elemento do loop interno, exatamente como descrito no exemplo anterior.

A versão abaixo procura apresentar no formulário as mudanças ocorridas no vetor, através do deslocamento dos elementos dentro dele:

```
public void InsertionSort()
{
    if (animar)
    {
        dgv.Parent.Height = 165 + 22;
        dgv.Rows.Add(); // usado para organizar o DataGridView para exibição
    }

    int inner, temp, linha = 0;
    for (int outer = 1; outer < numElements; outer++)
    {
        temp = arr[outer];
        inner = outer;
        if (animar)
        {
            dgv.Parent.Height += 22;
            linha = dgv.Rows.Add();
            for (int indice = 0; indice < numElements; indice++)
                dgv.Rows[linha - 1].Cells[indice].Value = Convert.ToString(arr[indice]);
        }

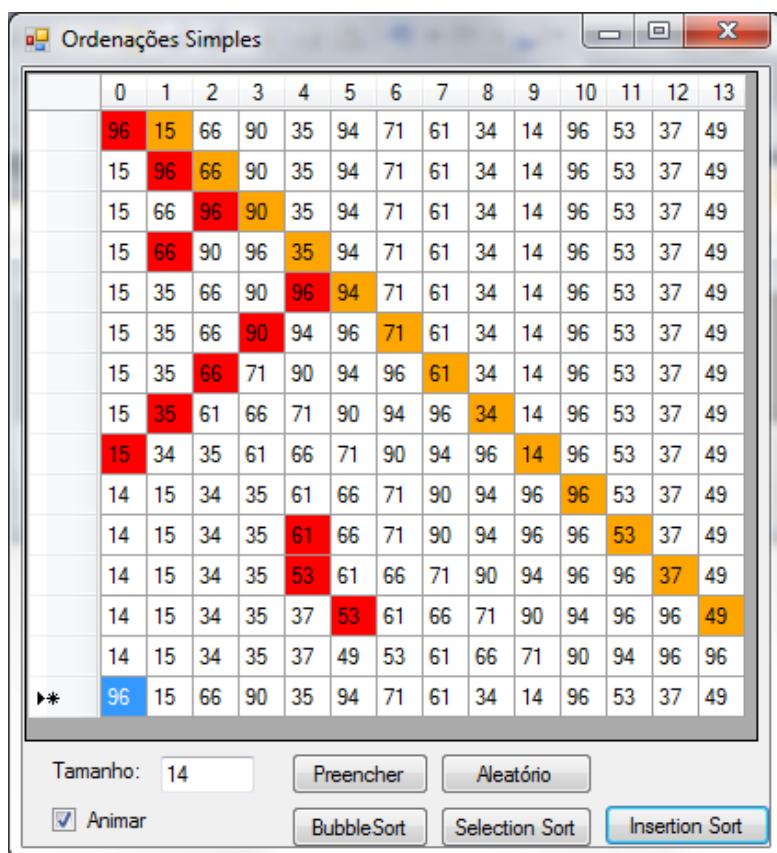
        while (inner > 0 && arr[inner - 1] > temp)
        {
            arr[inner] = arr[inner - 1];
            inner -= 1;
        }
        arr[inner] = temp;

        if (animar)
        {
            dgv.Rows[linha - 1].Cells[inner].Style.BackColor = Color.Red;
            dgv.Rows[linha - 1].Cells[outer].Style.BackColor = Color.Orange;
            Application.DoEvents();
        }
    }

    if (animar)
    {
        dgv.Parent.Height += 22;
        linha = dgv.Rows.Add();
        for (int indice = 0; indice < numElements; indice++)
            dgv.Rows[linha].Cells[indice].Value = Convert.ToString(arr[indice]);
    }
}
```

Executando, podemos observar como o Insertion Sort funciona com o conjunto de valores dos exemplos anteriores:

A figura ao lado mostra claramente que o Insertion Sort funciona não fazendo trocas, mas sim movendo elementos maiores do vetor para a direita, em busca de espaço para elementos menores que devem ser inseridos no lado esquerdo do vetor.



5. Comparação de Tempos entre Algoritmos de Ordenação Básicos

Estes três algoritmos de ordenação são bastante similares em complexidade e, teoricamente, ao menos, deveriam ter desempenho semelhante quando comparados uns com os outros. Podemos usar a classe Cronometro, descrita abaixo, para comparar os três algoritmos e ver se algum deles se destaca em relação aos demais em termos do tempo que ele leva para ordenar um grande conjunto de valores.

Para realizar o teste, usaremos o mesmo código básico que usamos anteriormente para demonstrar como cada algoritmo funciona. Nos testes a seguir, no entanto, o tamanho do vetor é variado para demonstrar como os três algoritmos se comportam tanto com conjuntos de dados pequenos quanto grandes. Os testes de tempo serão executados para vetores com tamanho de 100, 1.000 e 10.000 elementos. mas, antes disso, vamos discutir o código da classe Cronometro:

Uma classe para Testes de Tempo

Embora não seja necessária uma classe para testar o tempo de execução dos nossos algoritmos, faz sentido reescrever o código de teste de tempo como uma classe, primariamente porque com isso manteremos nosso código de implementação mais claro e mais coeso.

A classe Cronometro precisa dos seguintes membros de dados:

- tempolnicial — para armazenar o tempo inicial do código que estamos testando
- duracao — o tempo final do código que estamos testando

Os membros tempolnicial e duracao armazenam tempos e vamos usar o tipo de dados TimeSpan para esses campos. Teremos apenas um método construtor, um construtor default que atribui 0 aos dois campos.

Precisaremos de métodos para informar ao objeto da classe Cronometro quando iniciar a cronometragem do código sob teste e quando parar de cronometrar. Também precisaremos de um método para retornar o dado armazenado no campo duracao.

Como você pode ver, a classe Cronometro é bem pequena, precisando de apenas alguns métodos. Eis sua definição:

```
using System;
using System.Diagnostics;

namespace OrdenacoesSimples
{
    public class Cronometro
    {
        DateTime tempoInicial;
        DateTime duracao;

        public Cronometro()
        {
            tempoInicial = DateTime.Now;
            duracao      = tempoInicial;
        }

        public void pararCronometro()
        {
            duracao = DateTime.Now;
        }

        public void iniciarCronometro()
        {
            //GC.Collect();
            //GC.WaitForPendingFinalizers();
            tempoInicial = DateTime.Now;
        }

        public TimeSpan Result()
        {
            return duracao.Subtract(tempoInicial);
        }
    }
}
```

Vamos mudar os eventos onClick de cada botão de ordenação, para que eles passem a exibir o tempo gasto no processo:

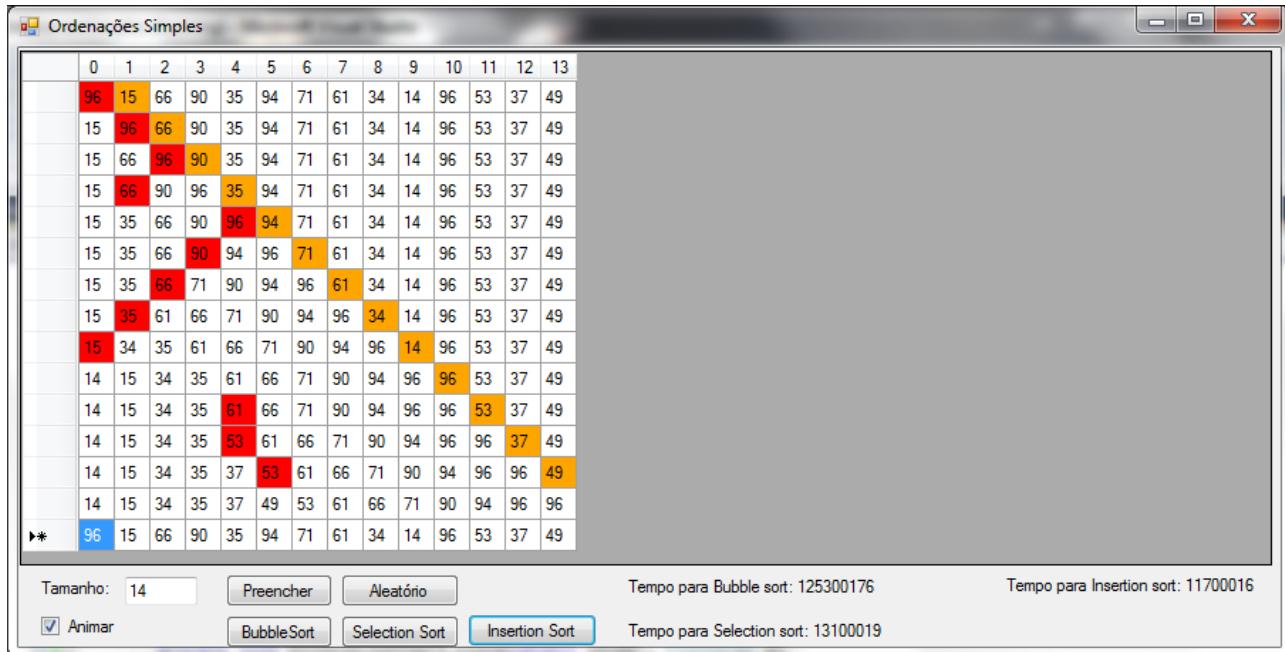
```
private void btnBubbleSort_Click(object sender, EventArgs e)
{
    Cronometro sortTime = new Cronometro();
    sortTime.iniciarCronometro();
    nums.BubbleSort();
    sortTime.pararCronometro();
    lblBubble.Text = "Tempo para Bubble sort: " + Convert.ToInt32(sortTime.Result().Ticks);
}

private void btnSelectionSort_Click(object sender, EventArgs e)
{
    Cronometro sortTime = new Cronometro();
    sortTime.iniciarCronometro();
    nums.SelectionSort();
    sortTime.pararCronometro();
    lblSelection.Text = "Tempo para Selection sort:" + Convert.ToInt32(sortTime.Result().Ticks);
}

private void btnSelecionSort_Click(object sender, EventArgs e)
{
    Cronometro sortTime = new Cronometro();
    sortTime.iniciarCronometro();
    nums.InsertionSort();
    sortTime.pararCronometro();
```

```
lblInsertion.Text="Tempo para Insertion sort:"+ Convert.ToInt32(sortTime.Result().Ticks);
}
```

A saída para 14 valores, com animação e após a execução das 3 ordenações, é:



Bubble: 125300176

Selection: 13100019

Insertion: 11700016

Sem animação, para 14 valores a ordenação é tão rápida que nosso programa não consegue medir a diferença de tempo.

Para 10.000 valores, sem animação:

```
Bubble      : 8400012
Selection   : 3600005
Insertion   : 2800004
```

Para 20.000 valores, sem animação:

```
Bubble      : 33400047
Selection   : 14400021
Insertion   : 11300016
```

Notamos que o tamanho do vetor faz uma grande diferença no desempenho dos algoritmos. O Selection Sort é cerca de 2,3 vezes mais rápido que o BubbleSort.

O desempenho dos três algoritmos degrada consideravelmente conforme aumenta o número de elementos. No entanto, observamos que o Bubble sort é o pior dos três. Claramente nenhum deles é adequado para a ordenação de grandes conjuntos de dados.

Vamos, portanto, examinar algoritmos para ordenar dados que são mais complexos do que os que aprendemos anteriormente. Esses algoritmos são, também, mais eficientes, e um deles, o algoritmo QuickSort, é geralmente considerado a ordenação mais eficiente para usar na maioria das situações. Os outros algoritmos que examinaremos são ShellSort, MergeSort e HeapSort.

Para comparar esses algoritmos avançados de ordenação, primeiramente discutiremos como cada um deles é implementado, e nos exercícios você usará a classe Cronometro para determinar a eficiência desses algoritmos.

6. O Algoritmo ShellSort

O algoritmo ShellSort é assim chamado devido ao seu inventor Donald Shell. Este algoritmo é, fundamentalmente, um aprimoramento do insertion sort. O conceito chave neste algoritmo é que ele compara itens que são distantes ao invés de comparar itens adjacentes, como se faz no insertion sort. Conforme o algoritmo percorre o conjunto de dados, a distância entre cada item diminui até que, ao final, o algoritmo estará comparando itens que são adjacentes.

ShellSort ordena elementos distantes pelo uso de uma sequência de incremento. A sequência deve começar com 1, mas pode então ser incrementada por qualquer quantidade. Um bom incremento para usar é baseado no fragmento de código abaixo:

```
while (h <= numElementos / 3)
    h = h * 3 + 1;
```

onde numElementos é o número de elementos no conjunto de dados sob ordenação, tal como um vetor.

Por exemplo, se o número de sequência gerado pelo código é 4, cada quarto elemento do conjunto de dados é ordenado. Em seguida, um novo número de sequência é escolhido, usando esse código:

```
h = (h - 1) / 3;
```

Então os próximos h elementos são ordenados, e assim por diante.

Vejamos o código para o algoritmo ShellSort:

```
public void ShellSort()
{
    int inner, temp;
    int h = 1;
    while (h <= numElements / 3)
        h = h * 3 + 1;
    while (h > 0)
    {
        for(int outer = h; outer < numElements; outer++)
        {
            temp = arr[outer];
            inner = outer;
            while ((inner > h-1) && arr[inner-h] >= temp)
            {
                arr[inner] = arr[inner-h];
                inner -= h;
            }
            arr[inner] = temp;
        }
        h = (h-1) / 3;
    }
}
```

Segue o código para testar o algoritmo. Abaixo temos, portanto, a versão para a classe CArray:

```
public void ShellSort()
{
    if (animar)
    {
```

```
dgv.Parent.Height = 165 + 22;
dgv.Rows.Add();           // usado para organizar o DataGridView para exibição
}

int inner, temp, linha=0;
int h = 1;
while (h <= numElements / 3)
    h = h * 3 + 1;
while (h > 0)
{
    if (animar)
    {
        dgv.Parent.Height += 22;
        linha = dgv.Rows.Add();
        for (int indice = 0; indice < numElements; indice++)
            dgv.Rows[linha-1].Cells[indice].Value=Convert.ToString(arr[indice]);
    }

    for (int outer = h; outer <= numElements - 1; outer++)
    {
        temp = arr[outer];
        inner = outer;
        while ((inner > h - 1) && arr[inner - h] >= temp)
        {
            arr[inner] = arr[inner - h];
            inner -= h;
            if (animar)
            {
                dgv.Rows[linha - 1].Cells[inner].Style.BackColor = Color.Red;
                dgv.Rows[linha - 1].Cells[outer].Style.BackColor = Color.Orange;
                Application.DoEvents();
            }
        }
        arr[inner] = temp;

        if (animar)
        {
            dgv.Rows[linha - 1].Cells[inner].Style.BackColor = Color.Red;
            dgv.Rows[linha - 1].Cells[outer].Style.BackColor = Color.Orange;
            Application.DoEvents();
        }
    }

    h = (h - 1) / 3;
}

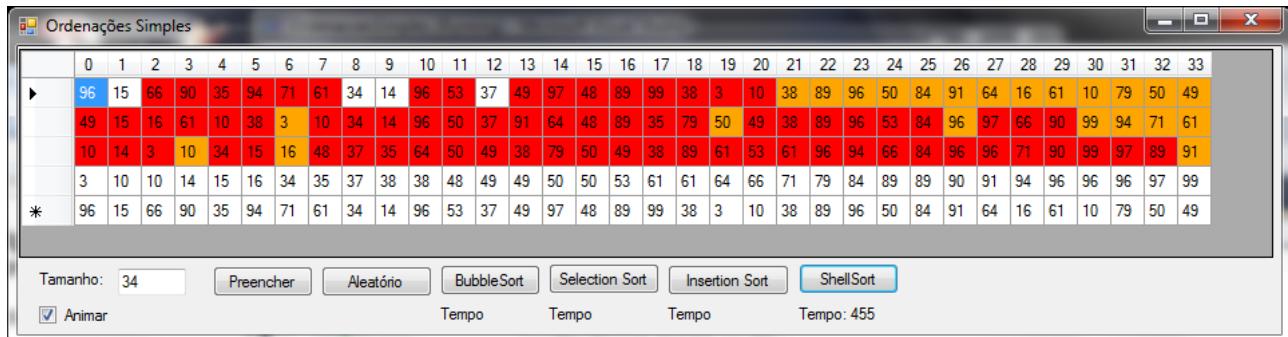
if (animar)
{
    dgv.Parent.Height = 165 + 22;
    dgv.Rows.Add();           // usado para organizar o DataGridView para exibição
    for (int indice = 0; indice < numElements; indice++)
        dgv.Rows[linha].Cells[indice].Value = Convert.ToString(arr[indice]);
}
}
```

No formulário, incluímos um botão btnShellSort e um label lblShell, como vemos na figura abaixo. O evento onClick do botão segue no código abaixo:

```
private void btnShellSort_Click(object sender, EventArgs e)
{
    Cronometro sortTime = new Cronometro();
    sortTime.iniciarCronometro();
```

```
        nums.ShellSort();
        sortTime.pararCronometro();
        lblShell.Text = "Tempo: " + Convert.ToInt32(sortTime.Result().Ticks);
    }
```

A figura abaixo mostra o resultado para 34 valores aleatoriamente distribuídos:



O ShellSort é frequentemente considerado um bom algoritmo avançado de ordenação para usar, porque ele é relativamente fácil de implementar mas seu desempenho é aceitável mesmo para conjuntos de dados contendo dezenas de milhares de elementos.

7. O Algoritmo de Ordenação por Casamento (MergeSort)

O algoritmo MergeSort é um bom exemplo de um algoritmo recursivo. Ele funciona "quebrando" o conjunto de dados em duas metades e recursivamente ordenando cada uma delas. Quando as duas metades estão ordenadas, elas são juntadas usando uma rotina de casamento de vetores (merge).

A parte fácil consiste na ordenação do conjunto de dados. Digamos que temos os seguintes dados no conjunto: 71 54 58 29 31 78 2 77. Primeiramente, o conjunto de dados é quebrado em dois conjuntos separados: 71 54 58 29 e 31 78 2 77. Em seguida, cada metade é ordenada: 29 54 58 71 e 2 31 77 78. Então os dois conjuntos são intercalados (casados) : 2 29 31 54 58 71 77 78.

O processo de casamento compara os dois primeiros elementos de cada conjunto de dados (armazenados em vetores temporários), copiando o menor valor para um terceiro vetor. O elemento que não foi adicionado ao terceiro vetor é então comparado ao próximo elemento do outro vetor. O menor dos dois elementos é adicionado ao terceiro vetor, e o processo continua até que os dois vetores tenham sido percorridos.

Mas e se um vetor é percorrido completamente antes do outro? Isso pode acontecer e o algoritmo toma precauções para esta situação. Dois loops extra são usados, de forma que cada um é executado apenas se um ou outro dos dois vetores ainda possuem dados não percorridos depois que o loop principal terminou.

Vejamos agora o código para realizar um MergeSort. Os dois primeiros métodos são o MergeSort e o recMergeSort. O primeiro simplesmente lança o método recursivo recMergeSort, que realiza a ordenação do vetor:

```
public void MergeSort()
{
    if (animar) {
        dgv.Parent.Height = 165 + 22;
        dgv.Rows.Add();           // usado para organizar o DataGridView para exibição
    }

    int[] tempArray = new int[numElements];
    RecMergeSort(tempArray, 0, numElements - 1);
```

```
if (animar) {
    dgv.Parent.Height += 22;
    int linha = dgv.Rows.Add();
    for (int indice = 0; indice < numElements; indice++)
        dgv.Rows[linha - 1].Cells[indice].Value = Convert.ToString(arr[indice]);
    Application.DoEvents();
}
}

public void RecMergeSort(int[] tempArray, int esquerdo, int direito)
{
    int linha = 0;
    if (esquerdo == direito)
        return;
    else
    {
        int meio = (int)(esquerdo + direito) / 2;

        if (animar) {
            dgv.Parent.Height += 22;
            linha = dgv.Rows.Add();
            for (int indice = 0; indice < numElements; indice++)
                dgv.Rows[linha-1].Cells[indice].Value=Convert.ToString(arr[indice]);
            dgv.Rows[linha - 1].Cells[esquerdo].Style.BackColor = Color.Red;
            dgv.Rows[linha - 1].Cells[meio + 1].Style.BackColor = Color.Green;
            dgv.Rows[linha - 1].Cells[direito].Style.BackColor = Color.Orange;
            Application.DoEvents();
        }

        RecMergeSort(tempArray, esquerdo, meio);
        RecMergeSort(tempArray, meio + 1, direito);
        Merge(tempArray, esquerdo, meio + 1, direito);
    }
}
```

Em RecMergeSort, o primeiro comando `if` é o caso básico da recursão, retornando ao local da chamada quando a condição fica verdadeira. Caso seja falsa, o ponto médio do trecho do vetor determinado pelos índices esquerdo e direito é calculado e a rotina é chamada recursivamente para a metade esquerda do trecho do vetor (a primeira chamada a RecMergeSort) e então para a metade direita do trecho do vetor (a segunda chamada a RecMergeSort). Finalmente, o trecho todo é casado pela chamada do método Merge, cujo código segue abaixo:

```
public void Merge(int[] tempArray, int lowp, int highp, int ubound)
{
    int lbound = lowp;
    int mid = highp - 1;
    int j = lowp;
    int n = (ubound-lbound) + 1;
    while ((lowp <= mid) && (highp <= ubound))
        if (arr[lowp] < arr[highp])
            tempArray[j++] = arr[lowp++];
        else
            tempArray[j++] = arr[highp++];

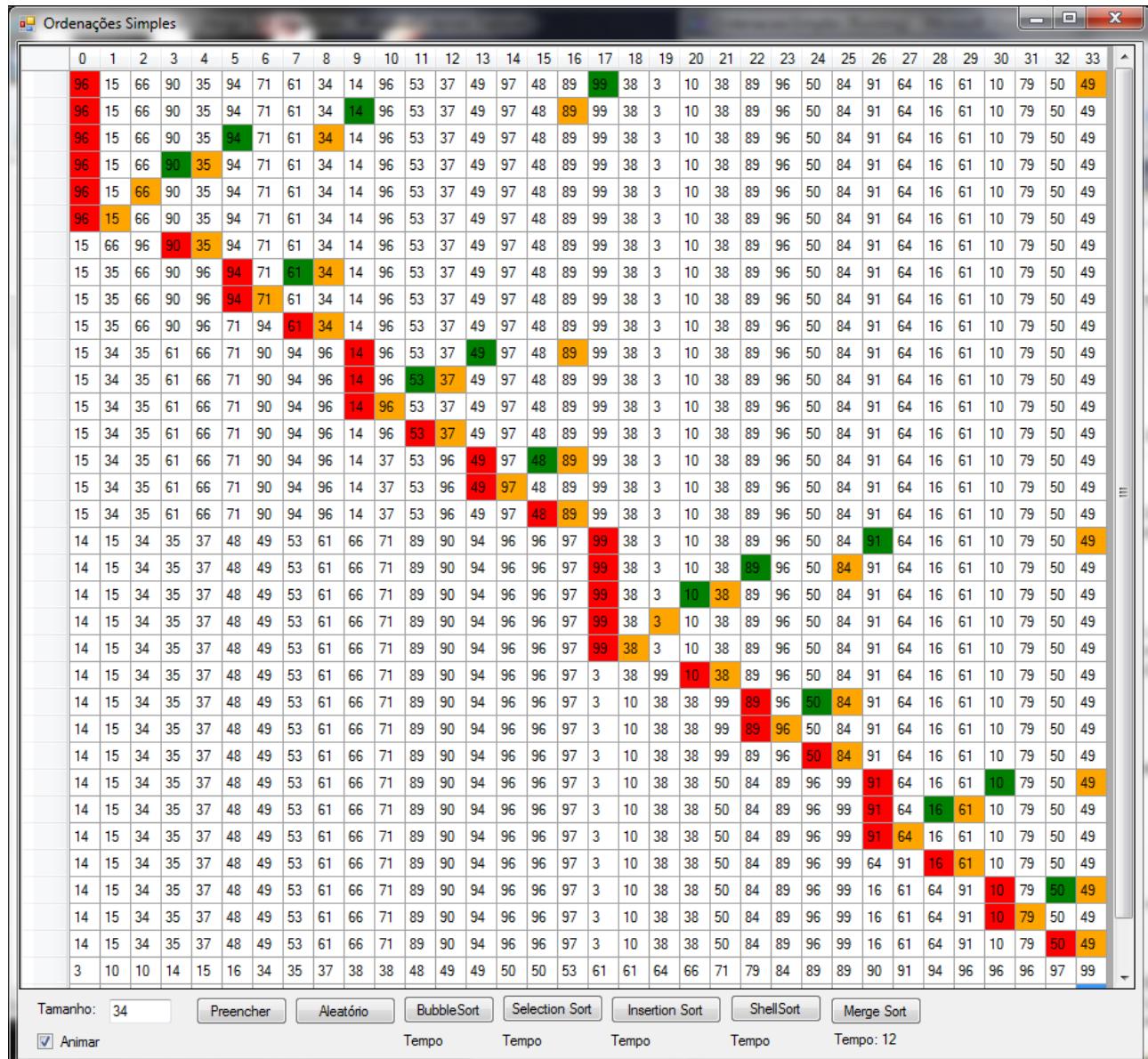
    while (lowp <= mid)
        tempArray[j++] = arr[lowp++];

    while (highp <= ubound)
        tempArray[j++] = arr[highp++];

    for(int i = 0; i <= n-1; i++)
        arr[lbound+i] = tempArray[lbound+i];
}
```

Esse método é chamado a cada vez que o método recMergeSort realiza uma ordenação preliminar. Na figura acima podemos ver o vetor em cada um dos estados temporários antes que ele seja completamente ordenado.

A primeira linha mostra o vetor no estado original, para 34 elementos aleatoriamente distribuídos. A segunda linha mostra o começo da metade esquerda sendo ordenada. Ao redor da metade da figura na vertical, começa a ordenação da parte direita. A última linha mostra o vetor ordenado.



8. O Algoritmo QuickSort

QuickSort tem uma reputação, obtida honestamente, como o mais rápido dos algoritmos de ordenação que discutiremos. Isto é verdade somente para conjuntos de dados muito grandes e em sua maioria sem ordenação. Se o conjunto de dados é pequeno (100 elementos ou menos), ou se os dados estão relativamente pré-ordenados, você deve usar um dos algoritmos iniciais e mais simples que já discutimos.

Descrição do Algoritmo QuickSort

Para entender como o algoritmo QuickSort funciona, imagine que você é um professor e que você tem que ordenar alfabeticamente uma pilha de provas de seus alunos. Você escolherá uma letra que está no meio do alfabeto, tal como M, e colocará todas as provas dos alunos cujo nome começa com letras de A a M em uma pilha e os nomes começando de N a Z em outra pilha. Em seguida, você divide a pilha A–M em duas pilhas (A-G e H-M) e a pilha N-Z em duas outras pilhas (N-S, T-Z) usando a mesma técnica. Então você faz a mesma coisa novamente até que você tenha um conjunto de pequenas pilhas (A-C, D-F, . . . , X-Z) de dois ou três elementos que podem ser ordenados facilmente. Uma vez que as pequenas pilhas estejam ordenadas, você simplesmente coloca todas as pilhas juntas e, assim, terá um conjunto completo de provas ordenadas.

Como você pode ter percebido, este processo é recursivo, desde que cada pilha é quebrada em pilhas mais e mais pequenas. Quando uma pilha é dividida em um elemento, essa pilha não pode ser mais dividida e a recursão para.

Como nós decidimos onde dividir o vetor em duas metades? Há muitas possíveis escolhas, mas nós começaremos justamente selecionando o primeiro elemento do vetor:

```
mv = arr[first];
```

Uma vez que essa escolha tenha sido feita, em seguida temos que entender como colocar os elementos do vetor na "metade" adequada do vetor. (A razão de a palavra metade estar entre aspas na frase anterior é que é inteiramente possível que as duas metades não sejam iguais, dependendo do ponto de separação das partições).

Conseguiremos isso criando duas variáveis, **inicio** e **fim**, armazenando o segundo índice (1) na variável **inicio** e o último índice na variável **fim**. Também criaremos uma outra variável, chamada **oPrimeiro**, que armazenará o primeiro elemento do vetor e que será considerado o **valor de separação**.

a.	<table border="1"> <tr> <td>87</td><td>91</td><td>65</td><td>72</td><td>84</td><td>99</td><td>89</td></tr> </table>	87	91	65	72	84	99	89	oPrimeiro	inicio	fim
87	91	65	72	84	99	89					
	valor de separação: 87										

① Incremente **inicio** até que $\text{arr}[\text{inicio}] > \text{valor de separação}$
inicio para no valor 91 (figura a).

② Decremente **fim** até que $\text{arr}[\text{fim}] \leq \text{valor de separação}$

<table border="1"> <tr> <td>87</td><td>91</td><td>65</td><td>72</td><td>84</td><td>99</td><td>89</td></tr> </table>	87	91	65	72	84	99	89	oPrimeiro	inicio	fim
87	91	65	72	84	99	89				

③ Troque os elementos $\text{arr}[\text{inicio}]$ e $\text{arr}[\text{fim}]$

<table border="1"> <tr> <td>87</td><td>84</td><td>65</td><td>72</td><td>91</td><td>99</td><td>89</td></tr> </table>	87	84	65	72	91	99	89	oPrimeiro	inicio	fim
87	84	65	72	91	99	89				

④ Incremente **inicio** até que $\text{arr}[\text{inicio}] > \text{valor de separação}$ ou $\text{inicio} > \text{fim}$
Decremente **fim** até que $\text{arr}[\text{fim}] \leq \text{valor de separação}$ ou $\text{fim} < \text{inicio}$

<table border="1"> <tr> <td>87</td><td>84</td><td>65</td><td>72</td><td>91</td><td>99</td><td>89</td></tr> </table>	87	84	65	72	91	99	89	oPrimeiro	inicio	fim
87	84	65	72	91	99	89				

⑤ **fim** está antes de **inicio** (ou **inicio** após **fim**)
Assim, troque elementos em **oPrimeiro** e **fim**

<table border="1"> <tr> <td>65</td><td>84</td><td>87</td><td>72</td><td>91</td><td>99</td><td>89</td></tr> </table>	65	84	87	72	91	99	89
65	84	87	72	91	99	89	

⑥ Repita o processo...

Código para o Algoritmo QuickSort

```
public void QSort()
{
    RecQSort(0, numElements - 1);
    if (animar)
    {
        dgv.Parent.Height += 22;
        int linha = dgv.Rows.Add();
        for (int indice = 0; indice < numElements; indice++)
            dgv.Rows[linha].Cells[indice].Value = Convert.ToString(arr[indice]);
        Application.DoEvents();
    }
}

public void RecQSort(int inicio, int fim)
{
    if (fim <= inicio)
        return;
    else
    {
        int pivot = arr[fim];
        int part = this.Partition(inicio, fim);
        RecQSort(inicio, part-1);
        RecQSort(part+1, fim);
    }
}

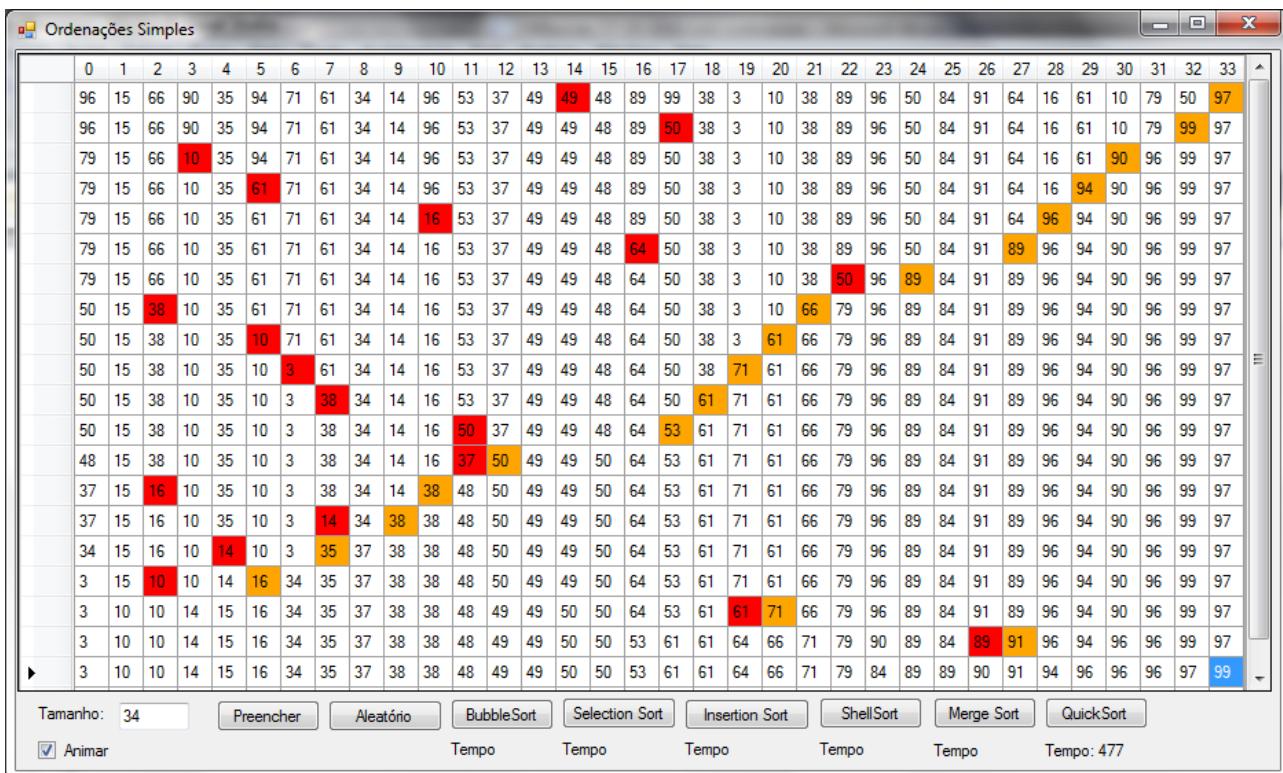
public int Partition(int inicio, int fim)
{
    int pivotVal = arr[inicio];
    int oPrimeiro = inicio;
    bool okSide;
    inicio++;
    do
    {
        okSide = true;
        while (okSide) // 4
            if (arr[inicio] > pivotVal)
                okSide = false;
            else
            {
                inicio++;
                okSide = (inicio <= fim);
            }
        okSide = (inicio <= fim);
        while (okSide) // 4
            if (arr[fim] <= pivotVal)
                okSide = false;
            else
            {
                fim--;
                okSide = (inicio <= fim);
            }
        if (inicio < fim)
        {
            Trocar(inicio, fim);
            if (animar)
            {
                dgv.Parent.Height += 22;
                int linha = dgv.Rows.Add();
                for (int indice = 0; indice < numElements; indice++)
                    dgv.Rows[linha].Cells[indice].Value =
            }
        }
    }
}
```

```

        Convert.ToString(arr[indice]));
        dgv.Rows[linha].Cells[inicio].Style.BackColor = Color.Red;
        dgv.Rows[linha].Cells[fim].Style.BackColor = Color.Orange;
        Application.DoEvents());
    }
    inicio++;
    fim--;
}
} while (inicio <= fim);
Trocar(oPrimeiro, fim);
return fim;
}

public void Trocar(int item1, int item2)
{
    int temp = arr[item1];
    arr[item1] = arr[item2];
    arr[item2] = temp;
}

```



Um aprimoramento no Algoritmo QuickSort

Se os dados no vetor são aleatórios, então selecionar o primeiro elemento como o pivô ou valor de separação (valor de partição) é perfeitamente aceitável. Se os dados não forem aleatórios, no entanto, fazendo esta escolha você inibirá o desempenho do algoritmo.

Uma escolha para selecionar este valor é determinar o valor mediano no vetor. Você pode fazer isso calculando a metade do limite superior de indexação do vetor. Por exemplo:

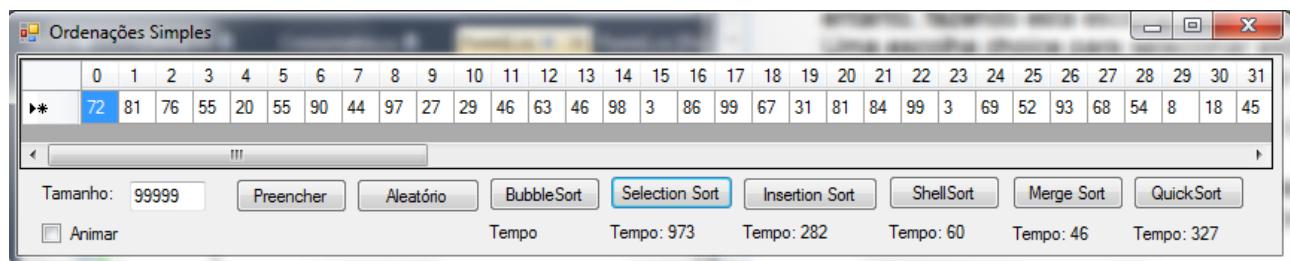
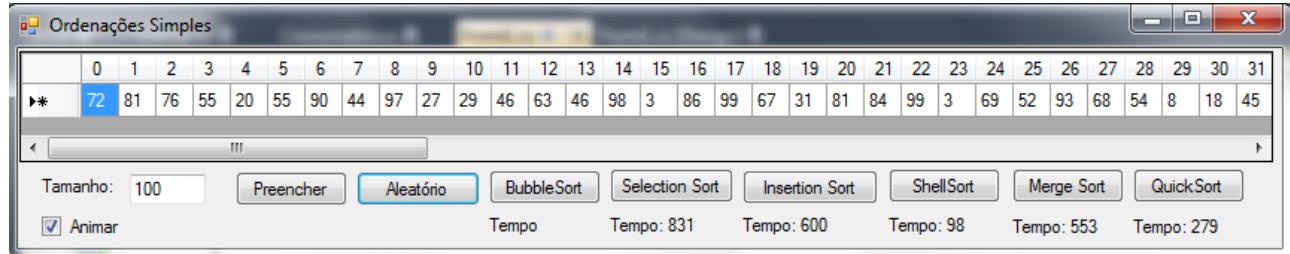
```
oPrimeiro = arr[(int)arr.GetUpperBound(0) / 2]
```

Estudos demonstraram que o uso dessa estratégia pode reduzir o tempo de execução do algoritmo em cerca de cinco por cento (veja Weiss 1999, p. 243).

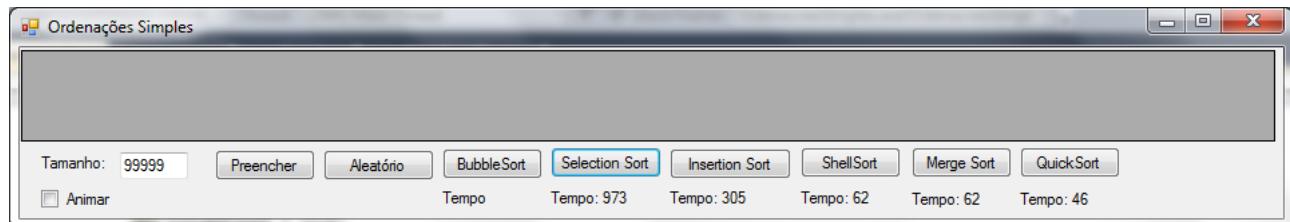
Outra fonte de informação: <http://www.publicjoe.f9.co.uk/csharp/sort00.html>

Comparação entre os vários métodos

Usando diversos tamanhos de vetores, com e sem animação, temos comparações abaixo:



Abaixo temos as ordenações e seus tempos, com os comandos que verificam se deve ou não haver animação no DataGridView comentados, de forma que mesmo o comando If que faz a verificação não é executado. Observe a mudança de velocidade do QuickSort para 99999 elementos:



9. O Algoritmo HeapSort

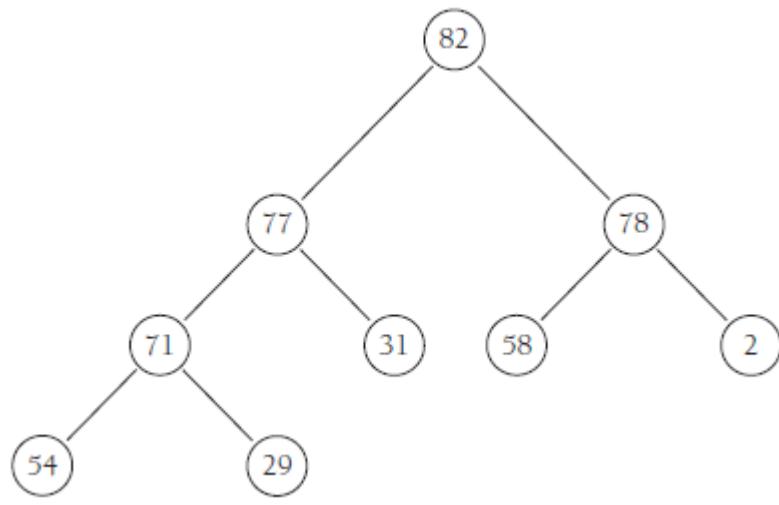
O algoritmo HeapSort usa uma estrutura de dados chamada *heap*. Um heap é similar a uma árvore binária, mas com algumas diferenças importantes. O algoritmo Heap-Sort, embora não seja o mais rápido dos algoritmos de ordenação, tem algumas características atraentes que encorajam seu uso em certas situações.

Construindo um Heap

A estrutura de dados Heap, como citamos antes, é similar a uma árvore binária, mas não exatamente igual. Primeiramente, heaps são geralmente construídos com vetores ao invés de referências a nós. Além disso, há também duas condições importantes para um heap:

1. um heap deve ser completo, ou seja, cada linha (nível) deve estar preenchida (a menos do último nível);
2. cada elemento contém dados que são maiores ou iguais aos dados que estão armazenados nos nós filhos abaixo dele.

Um exemplo de um heap e do vetor que o armazena é mostrado na figura abaixo.



82	77	78	71	31	58	2	54	29
0	1	2	3	4	5	6	7	8

Os dados que armazenamos em um heap são construídos a partir de uma classe No, similar aos nós que usamos anteriormente. Esta classe No específica, neste momento, armazenará apenas uma parte dos dados: o valor de sua chave primária. Não precisaremos de referências para outros nós (pois não é uma árvore que estamos montando), mas é interessante usarmos uma classe para os dados de forma que possamos facilmente trocar o tipo de dados sendo armazenado no heap se precisarmos fazê-lo. A seguir temos o código da classe No:

```

namespace OrdenacoesSimples {
    public class No {
        public int data;
        public No(int key) {
            data = key;
        }
    }
}
  
```

Heaps são construídos pela inserção de nós no vetor de heap, cujos elementos são os nós do heap. Um novo nó sempre é colocado ao fim do vetor, em uma posição vazia do vetor. O problema é que, fazendo isso, provavelmente quebraremos a condição do heap porque o valor da chave dos dados do novo nó pode ser maior que a chave de alguns dos nós acima dele. Para restaurar o vetor para a condição adequada de heap, temos que deslocar o novo nó até que ele encontre a posição adequada no vetor.

Faremos isso com um método chamado ShiftUp, cujo código vem a seguir:

```
public void ShiftUp(int index)
{
    int pai = (int) (index - 1) / 2;
    No bottom = heapArray[index];           // guarda para usar na comparação e no final
    while ((index > 0) && (heapArray[pai].data < bottom.data))
    {
        heapArray[index] = heapArray[pai];
        index = pai;
        pai = (pai - 1) / 2;
    }
    heapArray[index] = bottom;
}
```

E aqui temos o código para o método Insert:

```
public bool Insert(int key)
{
    if (currSize == maxSize)
        return False;
    No newNode = new No(key);
    heapArray[currSize] = newNode;
    ShiftUp(currSize);
    currSize++;
    return true;
}
```

O novo nó é adicionado ao final do vetor. Isto imediatamente quebra a condição do heap, assim a posição correta do novo nó no vetor deverá ser encontrada pelo método ShiftUp. O argumento para este método é o índice do novo nó.

O pai deste nó é calculado na primeira linha do método. Em seguida, o novo nó é salvo na variável bottom, do tipo No. O loop while, em seguida, encontra a posição correta para o novo nó. A última linha, por sua vez, copia o novo nó de seu local temporário na variável bottom para sua posição correta no vetor.

Remover um nó de um heap sempre significa remover o nó com o maior valor. Isto é fácil de fazer porque o valor máximo sempre está no nó raiz. O problema é que, uma vez que o nó raiz seja removido, o heap ficará incompleto e deve ser reorganizado. Existe um algoritmo para tornar um heap completo novamente:

1. Remova o nó na raiz.
2. Mova o nó da última posição para a raiz.
3. Desça o último nó até que ele fique abaixo (TrickleDown --> gotejar para baixo).

Quando este algoritmo é aplicado continuamente, os dados são removidos do heap em ordem (classificados). Abaixo temos o código dos métodos Remove e ShiftDown:

```
public No Remove()
{
    No root = heapArray[0];
    currSize--;
    heapArray[0] = heapArray[currSize];
    ShiftDown(0);
}
```

```
        ShiftDown(0);
        return root;
    }

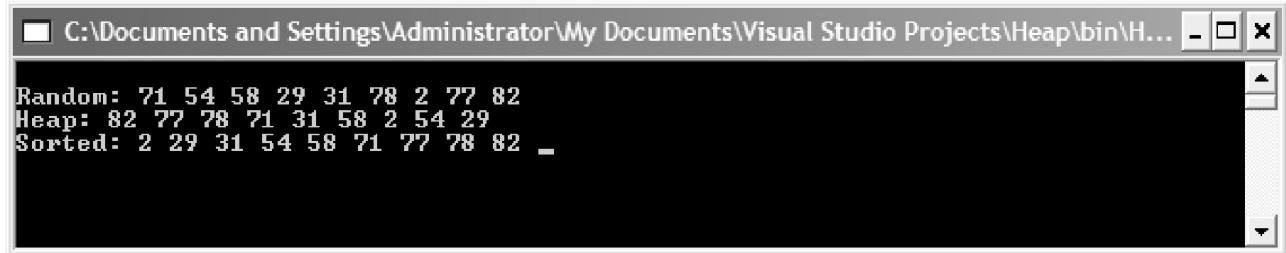
    public void ShiftDown(int index)
    {
        int largerChild;
        No top = heapArray[index];
        while (index < (int)(currSize / 2))
        {
            int leftChild = 2 * index + 1;
            int rightChild = leftChild + 1;
            if ((rightChild < currSize) &&
                heapArray[leftChild].data < heapArray[rightChild].data)
                largerChild = rightChild;
            else
                largerChild = leftChild;
            if (top.data >= heapArray[largerChild].data)
                break;
            heapArray[index] = heapArray[largerChild];
            index = largerChild;
        }
        heapArray[index] = top;
    }
```

Isto é tudo que precisamos para fazer um heap sort, portanto vamos mudar nosso formulário para que ele construa um heap e o ordene:

```
static void Main()
{
    const int SIZE = 9;
    Heap aHeap = new Heap(SIZE);
    No sortedHeap = new No[SIZE];
    for(int i = 0; i < SIZE; i++)
    {
        Random RandomClass = new Random();
        int rn = RandomClass.Next(1,100);
        No aNode = new No(rn);
        aHeap.InsertAt(i, aNode);
        aHeap.IncSize();
    }
    Console.Write("Random: ");
    aHeap.ShowArray();
    Console.WriteLine();
    Console.Write("Heap: ");
    for(int i = (int)SIZE/2-1; i >= 0; i--)
        aHeap.ShiftDown(i);
    aHeap.ShowArray();
    for(int i = SIZE-1; i >= 0; i--)
    {
        Node bigNode = aHeap.Remove();
        aHeap.InsertAt(i, bigNode);
    }
    Console.WriteLine();
    Console.Write("Sorted: ");
    aHeap.ShowArray();
}
```

O primeiro loop for inicia o processo de construção do heap através da inclusão de números aleatórios no heap. O segundo loop torna o heap correto e o terceiro, por sua vez, usa o método

Remove e o método ShiftDown para reconstruir o heap de forma ordenada. Eis a saída do programa em modo console:



```
Random: 71 54 58 29 31 78 2 77 82
Heap: 82 77 78 21 31 58 2 54 29
Sorted: 2 29 31 54 58 71 77 78 82 _
```

Mude o nosso formulário e a classe CArray para que eles implementem os métodos dessa ordenação e, também, a sua visualização no formulário.

HeapSort é o segundo mais rápido dos algoritmos avançados que examinamos neste capítulo. Somente o algoritmo Quicksort é mais rápido.

LaFore, Robert. *Data Structures and Algorithms in Java*, Corte Madera, CA : Waite Group Press, 1998.

Weiss, Mark Allen. *Data Structures and Algorithm Analysis in Java*, Reading, MA: Addison-Wesley, 1999.

```
-----the stars,-----
|   .-'          |  Francisco da Fonseca Rodrigues  |      o   o   | | | |
| ,_| \ ._/\     |  COTUCA-Colégio Técnico da UNICAMP |      o       o   |
| |  |o/^^~-_.-| ~|                                |      o           |
| /-'    BRASIL  |  | Depto de Processamento de Dados |      o   o   o   o   |
| \__/_|_        |  | e-mail: chico@unicamp.br      |      o   o   o   o   |
|   \_  Cps      |  | chico@cotuca.unicamp.br    |      o           |
|     | * /'      |  | Telefone : 55-19-3521-9954 |      o   o   |
|     /' /'      |  | Campinas - SP - Brasil      |      o   o   |
-----like dust.-----
```

APÊNDICE

Baseado e traduzido de

Data Structures and Algorithms with Object-Oriented Design Patterns in C# - Bruno Preiss (<http://www.brpreiss.com/books/opus6/html/book.html>)

1. Framework .Net

Introdução

A revolução da Internet no final dos anos 1990 representou uma mudança dramática na forma como indivíduos e organizações se comunicavam uns com os outros.

Aplicações tradicionais, tais como processadores de texto e pacotes de contabilidade são modelados como aplicações stand-alone: elas oferecem aos usuários a capacidade de executar tarefas usando dados armazenados no sistema em que a aplicação reside e onde ela é executada.

A maioria dos novos sistemas, por outro lado, é modelada baseada num modelo de computação distribuída onde as aplicações colaboram para fornecer serviços e expor funcionalidades umas às outras. Como resultado, o papel principal da maioria dos novos sistemas está sendo alterado para suportar intercâmbio de informações (através de servidores Web e browsers), colaboração (através de e-mail e programas de mensagens instantâneas) e a expressão indivisual (através de Web logs, também conhecidos como Blogs e e-zines – revistas baseadas em Web). Essencialmente, o papel básico do software está sendo alterado do fornecimento de funcionalidades discretas (indivisíveis) para o fornecimento de serviços.

O .NET Framework representa um conjunto unificado de serviços e bibliotecas orientados a objetos, adaptados ao papel, em mutação, dos novos softwares centrados em redes e conscientes de que estão sendo executados em redes. De fato, o .NET Framework é a primeira plataforma de software projetada, desde seus alicerces, tendo a Internet em mente.

É um conjunto de objetos e interfaces dos sistemas operacionais da Microsoft para construir aplicativos. Esses objetos e interfaces foram desenvolvidos para facilitar o trabalho de desenvolvimento de aplicativos que se comunicam entre si (distribuídos).

Benefícios do .NET Framework

O .NET Framework oferece vários benefícios aos desenvolvedores de software:

- Um modelo consistente de programação
- Suporte direto para segurança
- Esforços de desenvolvimento simplificados
- Fácil distribuição e manutenção das aplicações

Modelo consistente de programação

Linguagens de programação distintas oferecem modelos distintos para realizar as mesmas tarefas. Por exemplo, o código abaixo demonstra como abrir um arquivo e nele escrever uma mensagem de uma única linha usando Visual Basic 6.0:

```
Public Sub testFileAccess()
    On Error GoTo handle_Error

    ' Use native method of opening an writing to a file...
    Dim outputFile As Long
    outputFile = FreeFile
    Open "c:\temp\test.txt" For Output As #outputFile
    Print #outputFile, "Hello World!"
    Close #outputFile
```

```
' Use the Microsoft Scripting Runtime to
' open and write to the file...

Dim fso As Object
Set fso = CreateObject("Scripting.FileSystemObject")
Dim outputText As TextStream
Set outputText = fso.CreateTextFile("c:\temp\test2.txt")
outputText.WriteLine "Hello World!"
outputText.Close
Set fso = Nothing
Exit Sub

handle_Error:

    ' Handle or report error here
End Sub
```

O código acima demonstra que mais de uma técnica existe para criar e escrever em um arquivo. O primeiro método usa o suporte interno do próprio Visual Basic, enquanto o Segundo método usa o Microsoft Scripting Runtime. C++ também oferece mais de uma forma de realizar a mesma tarefa, como vemos no código a seguir:

```
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <stdio.h>
using namespace std;
int main(int argc, char* argv[])
{

// Use the C Runtime Library (CRT) ...

FILE *testFile;
if( (testFile = fopen( "c:\\temp\\\\test3.txt","wt" )) == NULL ) {
    cout << "Could not open first test file!" << endl;
    return 1;
}
fprintf(testFile,"Hello World!\n");
fclose(testFile);

// Use the Standard Template Library (STL) ...

ofstream outputStream("c:\\temp\\\\test4.txt");
if(!outputStream) {
    cout << "Could not open second test file!" << endl;
    return(1);
}
outputStream << "Hello World!" << endl;
outputStream.close();
return 0;
}
```

Ambos os códigos demonstram que, quando se usam diferentes linguagens de programação, existe uma disparidade entre as técnicas que os desenvolvedores utilizam para realizar a mesma tarefa.

A diferença nas técnicas provém de como as diferentes linguagens representam e interagem com o sistema subjacente ao qual as aplicações se referem, aumentando a quantidade de treinamento que os desenvolvedores necessitam para poderem usar várias linguagens.

Já o código a seguir demonstra como realizar as mesmas tarefas em Visual Basic .NET e Visual C# .NET:

Visual Basic .NET:

```
Imports System.IO
Imports System.Text
Module Demo
    Sub Main()
        Dim outputFile As StreamWriter = New StreamWriter("c:\temp\test5.txt")
        outputFile.WriteLine("Hello World!")
        outputFile.Close()
    End Sub
End Module
```

C# .NET:

```
using System.IO;
using System.Text;
class Demo {
    static void Main() {
        StreamWriter outputFile = new StreamWriter("c:\\temp\\test6.txt");
        outputFile.WriteLine("Hello World!");
        outputFile.Close();
    }
}
```

O código anterior demonstra, à parte as pequenas diferenças sintáticas, que a técnica para escrever em um arquivo nas duas linguagens é idêntico – ambas as listagens utilizam a classe StreamWriter para escrever a mensagem "Hello World!" no arquivo texto. Verdadeiramente, ao contrário das listagens escritas em Visual Basic e em Visual C++, que demonstram que há mais de uma maneira de fazer alguma coisa *dentro da mesma linguagem*, as listagens precedentes mostram que existe um meio uniforme de realizar a mesma tarefa usando a Biblioteca de Classes .NET (.NET Class Library).

A Biblioteca de Classes .NET Class Library é um componente chave do .NET Framework — ela é referenciada, algumas vezes, como a Biblioteca de Classes Base ou Base Class Library (BCL). A Biblioteca de Classes .NET contém centenas de classes que você pode usar para tarefas tais como as seguintes:

- Processamento de XML
- Trabalhar com dados de múltiplas fontes de dados
- Depuração de código e trabalho com log (diário) de eventos ocorridos durante a execução
- Trabalhar com arquivos e streams (correntes, fluxos) de dados
- Gerenciamento do ambiente de execução
- Desenvolvimento de serviços Web, componentes e aplicações padrão do Windows
- Trabalhar com segurança de aplicações
- Trabalhar com serviços de diretório

As funcionalidades que a Biblioteca de Classes .NET fornece estão disponíveis para todas as linguagens compatíveis com a plataforma .NET, o que resulta em um modelo de objetos consistente, independentemente da linguagem de programação que o desenvolvedor utiliza.

Suporte Direto à Segurança

Desenvolver uma aplicação que resida num sistema desktop de um usuário e utiliza recursos locais é fácil, do ponto de vista de segurança, porque a segurança simplesmente não é uma consideração neste cenário.

A segurança se torna muito mais importante quando você cria aplicações que acessam dados em sistemas remotos ou aplicações que realizam tarefas privilegiadas em nome de usuários não-privilegiados, porque os sistemas deverão autenticar usuários e poderá ser necessária criptografia para tornar seguras as comunicação de dados.

Windows NT, Windows 2000 e Windows XP possuem vários recursos de segurança, baseados em Listas de Controle de Acesso (Access Control Lists - ACLs). Uma ACL contém várias entradas que especificam quais usuários podem acessar, ou cujo acesso é explicitamente negado, aos recursos tais como arquivos e impressoras. Os ACLs são uma ótima maneira de proteger arquivos executáveis de acesso não-autorizado, mas elas não asseguram todas as partes do arquivo. O .NET Framework permite que tanto desenvolvedores quanto administradores de sistemas especifiquem segurança em nível de métodos dos objetos em execução.

Desenvolvedores (através de atributos - construtores de linguagens de programação, fáceis de usar,) e administradores de sistemas (usando ferramentas de administração e editando arquivos de configuração de aplicações) podem configurar a segurança de uma aplicação de forma que somente usuários autorizados possam invocar métodos internos de um componente ou objeto.

O .NET Framework utiliza protocolos padronizados pela indústria de computação, como o TCP/IP, e meios de comunicação tais como Extensible Markup Language (XML), Simple Object Access Protocol (SOAP, um protocolo padrão para envio e recebimentos de mensagens) e HTTP para facilitar a comunicação entre aplicações distribuídas. Isto torna a computação distribuída mais segura, porque desenvolvedores .NET cooperam com dispositivos de conectividade de rede ao invés de desenvolver estratégias de software para tentar superar as restrições de segurança.

Esforço de Desenvolvimento Simplificado

Dois aspectos da criação de aplicações baseadas em Web apresentam desafios únicos aos desenvolvedores Web: o projeto visual das páginas e a depuração de aplicações. O projeto visual das páginas é fácil e direto quando se cria conteúdo estático; no entanto, quando você necessita apresentar os resultados da execução de uma consulta a banco de dados em um formato tabular usando um script ASP, o projeto da página pode se tornar bastante difícil. Isso acontece porque os desenvolvedores precisam misturar código ASP tradicional, que representa a lógica da aplicação, e HTML, que representa a apresentação dos dados.

ASP.NET e o .NET Framework simplificam o desenvolvimento ao permitir que os desenvolvedores separem a lógica da aplicação de sua apresentação, resultando em um código mais fácil de manter e de projetar visualmente, sem a interferência de códigos programáveis no interior de código HTML.

ASP.NET também pode gerenciar os detalhes da manutenção do estado de controles, tais como os conteúdos de caixas de texto, entre chamadas para a mesma página ASP.NET, de forma a reduzir a quantidade de código que você precisa escrever.

O Ambiente de Desenvolvimento Visual Studio .NET, que é integrado ao .NET Framework, dá assistência aos desenvolvedores quando eles criam aplicações ASP.NET, fornecendo ferramentas visuais de design que facilitam a edição visual por drag and drop (comem Visual Basic e Delphi), tornando o layout de páginas e de formulários de dados muito mais fácil.

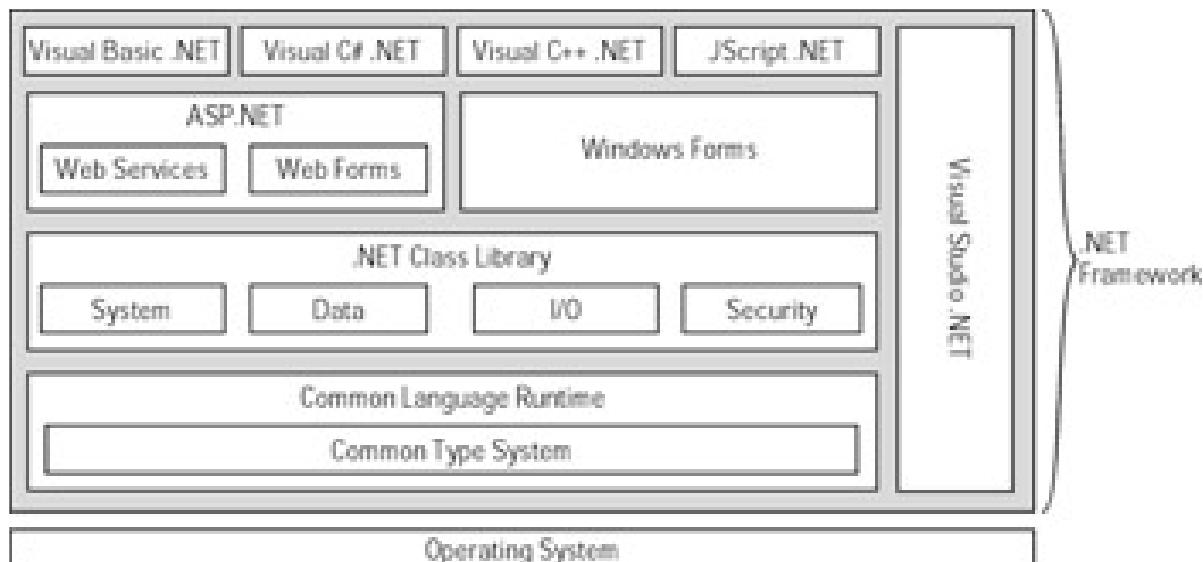
Outro aspecto da criação de aplicações é a depuração. Desenvolvedores, por vezes, cometem erros; sistemas não se comportam como seria esperado deles e condições inesperadas aparecem – todos esses problemas são, coletivamente, conhecidos como “bugs”. Rastrear os bugs – tarefa conhecida como “debugging” ou depuração – rápida e efetivamente exige que os desenvolvedores se familiarizem com uma variedade de ferramentas, algumas vezes de terceiros, e com técnicas – uma combinação de técnicas de programação e técnicas para usar com uma ferramenta

específica. O .NET Framework simplifica a depuração com suporte de diagnósticos de Runtime (tempo de execução).

Diagnósticos de Runtime não somente ajudam o desenvolvedor a rastrear os erros mas também a determinar quão bem uma aplicação se desempenha e assegurar as condições dessa aplicação. O .NET Framework fornece 3 tipos de Diagnósticos de Runtime:

- Registro de eventos (Event logging)
- Contadores de desempenho (Performance counters)
- Rastreio (Tracing)

Elementos do .NET Framework



O .NET Framework consiste de três elementos chaves, como vemos na figura acima:

- Common Language Runtime
- .NET Class Library
- Componentes Unificadores

Common Language Runtime

A CLR é um ambiente que gerencia a execução de código compilado de classes do .Net Framework. Um compilador compatível com .Net gera código intermediário (MSIL – Microsoft Intermediate Language) e não executável. Esse código intermediário é lido e executado pelo sistema operacional da máquina de execução. Após compilado o aplicativo, pode-se executá-lo em qualquer lugar (que possua CLR). O código executado pelo CLR é gerenciado, fornecendo recursos prontos para uso, como segurança, comunicação, controle de memória, etc.

A Common Language Runtime (CLR) é uma camada entre uma aplicação e o sistema operacional onde a aplicação é executada. Ela simplifica o projeto de uma aplicação e reduz a quantidade de código que o desenvolvedor precisa escrever, pois ela fornece uma grande variedade de serviços de execução que incluem gerenciamento de memória, gerenciamento de threads, gerenciamento do tempo de vida de componentes e tratamento de erros. O benefício maior da CLR é que ela fornece esses serviços transparentemente para todas as aplicações, independentemente de qual linguagem de programação foi usada para escrever a aplicação e sem nenhum esforço adicional por parte do desenvolvedor.

A CLR é também responsável pela compilação de código logo antes que ele seja executado. Ao invés de produzir uma representação binária do seu código, em linguagem de máquina, como os compiladores tradicionais fazem (plataforma Win32), compiladores compatíveis com .NET

Producem uma representação de seu código em uma linguagem **comum** do .NET

Framework: Microsoft Intermediate Language (MSIL), freqüentemente chamada de IL. Quando seu código é executado pela primeira vez, a CLR invoca um compilador especial chamado Just In Time (JIT) compiler, que transforma a IL em instruções executáveis específicas ao tipo e modelo do processador do seu computador. Devido ao fato de todas as linguagens compatíveis com .NET terem a mesma representação compilada, elas todas possuem características similares de desempenho. Isto significa que um programa escrito em Visual Basic .NET tem desempenho tão bom quanto o mesmo programa escrito em Visual C++ .NET. (e C++ é a linguagem de alto nível mais escolhida pelos desenvolvedores que necessitam do melhor desempenho que um sistema possa oferecer).

Common Type System

O Common Type System (CTS) é um componente da CLR e fornece um conjunto comum de tipos de dados, cada um tendo um conjunto comum de comportamentos. Em Visual Basic, por exemplo, o tipo de dados String é mapeado para a classe System.String do CTS. Portanto, se um cliente JScript

.NET precisa se comunicar com um componente implementado em VB.NET, o cliente não tem que fazer qualquer trabalho adicional para trocar informações, porque ele está usando um tipo comum tanto a JScript .NET quanto VB .NET. O CTS elimina muitos problemas de interoperabilidade que aconteciam com freqüência antes de existir a plataforma .NET.

As linguagens de programação compatíveis com .NET se beneficiam do CTS ao permitir que os desenvolvedores usem os tipos padrão já implantados em suas linguagens – os compiladores .NET convertem os tipos de dados nativos para os seus equivalentes aos tipos CTS em tempo de compilação. Os desenvolvedores podem também usar tipos CTS diretamente em seu código se assim desejarem.

Nome completo	Descrição
System.Byte	Inteiro de 8 bits sem sinal no intervalo de 0 a +255
System.Int16	Inteiro de 16 bits com sinal, no intervalo de -32.768 a +32.767
System.Int32	Inteiro de 32 bits com sinal, no intervalo de -2.147.483.648 a + 2.147.483.647
System.Int64	Inteiro de 64 bits com sinal, no intervalo de -9.223.372.036.854.755.808 a +9.223.372.036.854.755.807
System.Single	Número de ponto flutuante, de 32 bits, com precisão simples
System.Double	Número de ponto flutuante, de 64 bits, com precisão dupla
System.Decimal	Valor de 96 bits e ponto flutuante, com sinal, com até 28 dígitos em cada lado do ponto decimal
System.Char	Caracter Unicode de 16 bits (valores sem sinal)
System.String	Seqüência de caracteres Unicode com capacidade de até 2 bilhões de caracteres
System.Object	Endereço de 32 bits, referenciando uma instância de uma classe
System.Boolean	Número sem sinal, de 32 bits, que podem conter somente 0 (False) ou 1 (True)

Existem tipos adicionais não-compatíveis com CTS, que podem ser usados mas que poderão não funcionar em todos os ambientes operacionais:

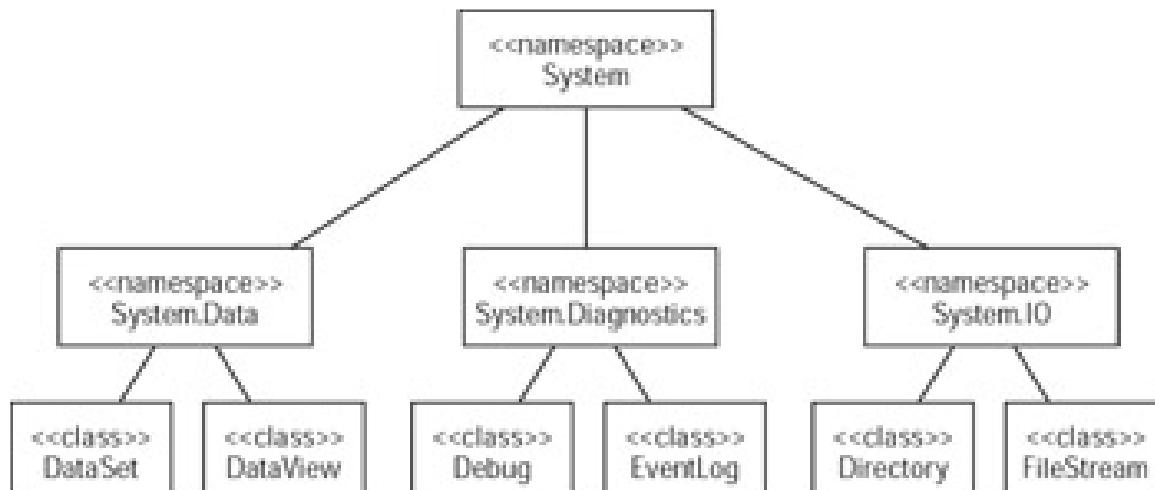
Nome completo	Descrição
System.SByte	Inteiro de 8 bits com sinal no intervalo de -128 a +127
System.UInt16	Inteiro de 16 bits sem sinal, no intervalo de 0 a +65.535

System.UInt32	Inteiro de 32 bits sem sinal, no intervalo de 0 a +4.294.967.295
System.UInt64	Inteiro de 64 bits sem sinal, no intervalo de 0 a +184.467.440.737.095.551.615

A .NET Class Library foi descrita anteriormente, como contendo centenas de classes que modelam o sistema operacional e os serviços que ele fornece. Para facilitar o trabalho com a Biblioteca de Classes .NET, ela é dividida em espaços de nomes, ou *namespaces*.

O namespace raiz da .NET Class Library é chamado System, e contém as classes de núcleo e tipos de dados, tais como Int32, Object, Array e Console. Namespaces secundários residem dentro do namespace System. Exemplos de namespaces aninhados incluem os seguintes:

- **System.Diagnostics**: contém classes para trabalhar com o Event Log
- **System.Data**: facilita o trabalho com dados de múltiplas fontes de (System.Data.OleDb reside neste namespace e contém as classes ADO.NET, que permitem acesso a bancos de dados e outros tipos de organização de informações)
- **System.IO**: contém classes para trabalhar com arquivos e fluxos de dados (streams)



Os benefícios de usar a .NET Class Library incluem um conjunto consistente de serviços disponíveis a todas as linguagens compatíveis com .NET e uma distribuição simplificada, pois a .NET Class Library está disponível em todas as implementações do .NET Framework.

Você pode obter descrições pormenorizadas e exemplos de código sobre as classes da Biblioteca de Classes .Net no site da Microsoft.

Componentes Unificadores

Até este momento, cobrimos os componentes de baixo nível do .NET Framework. Os componentes unificadores, listados abaixo, são os meios pelos quais você pode acessar os serviços fornecidos pelo .NET Framework:

- ASP.NET
- Windows Forms
- Visual Studio.Net

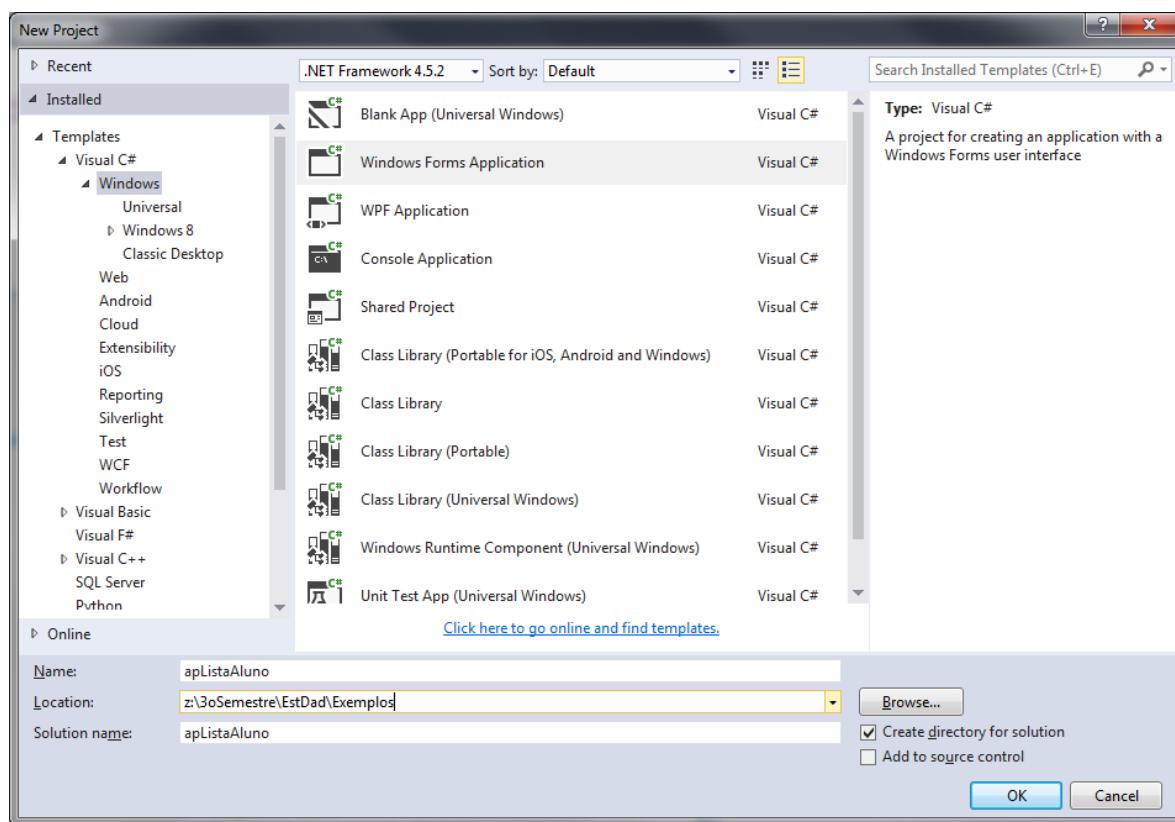
2. Linguagem C#

C# e Programação Orientada a Objetos

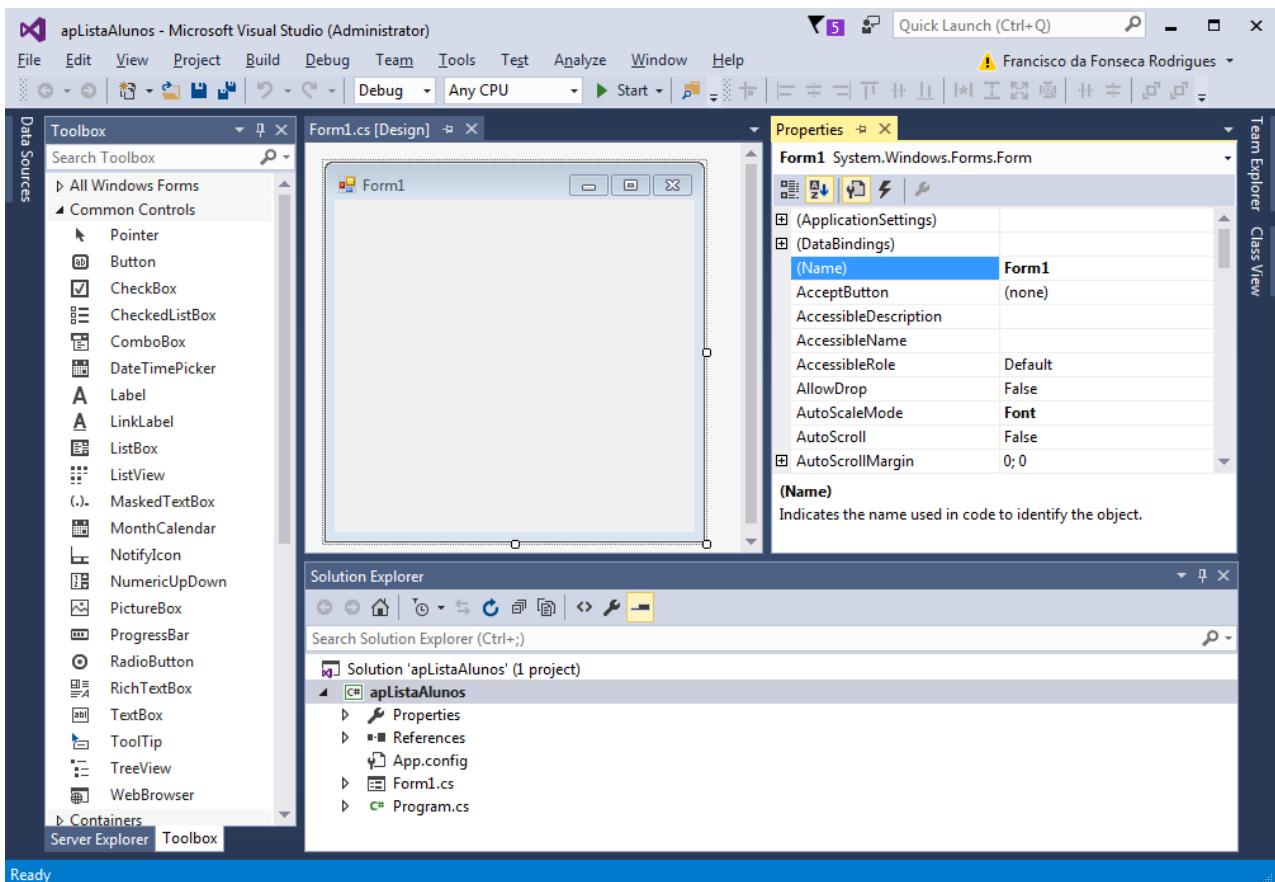
Para aprendermos C# vamos usar o ambiente de programação Visual Studio, que também permite programação em outras linguagens compatíveis com o .Net Framework, como Visual Basic e C++. Também se podem usar as versões Express do Visual Studio como, por exemplo, a Visual C# Express.

Quando você acessa a opção File | New Project, aparece a janela ao lado, que lhe permite indicar o tipo de projeto que criará, bem como informar o nome desse projeto:

Escolhendo Windows Forms Application, podemos criar formulários Windows como os que criamos em Delphi. Escolhendo Console Application, criamos aplicações console, para serem executadas na linha de comandos do Windows.



A seguir, vemos o resultado da criação de um projeto Windows Forms:



Observe a Toolbox, que contém os componentes que podem ser arrastados para o formulário.
Observe também o Solution Explorer, que contém uma lista dos arquivos do projeto.

Nessa janela, observe o item Form1 usado para o projeto visual do formulário e Program.cs, para colocação do código fonte em C#.

Clique duas vezes em Program.cs e o código já criado aparecerá.

Note que ele se parece bastante com Java, e que a sequência de comandos lembra um programa principal de Delphi.

As janelas do ambiente são muito semelhantes às do Turbo Delphi em aparência e utilização.

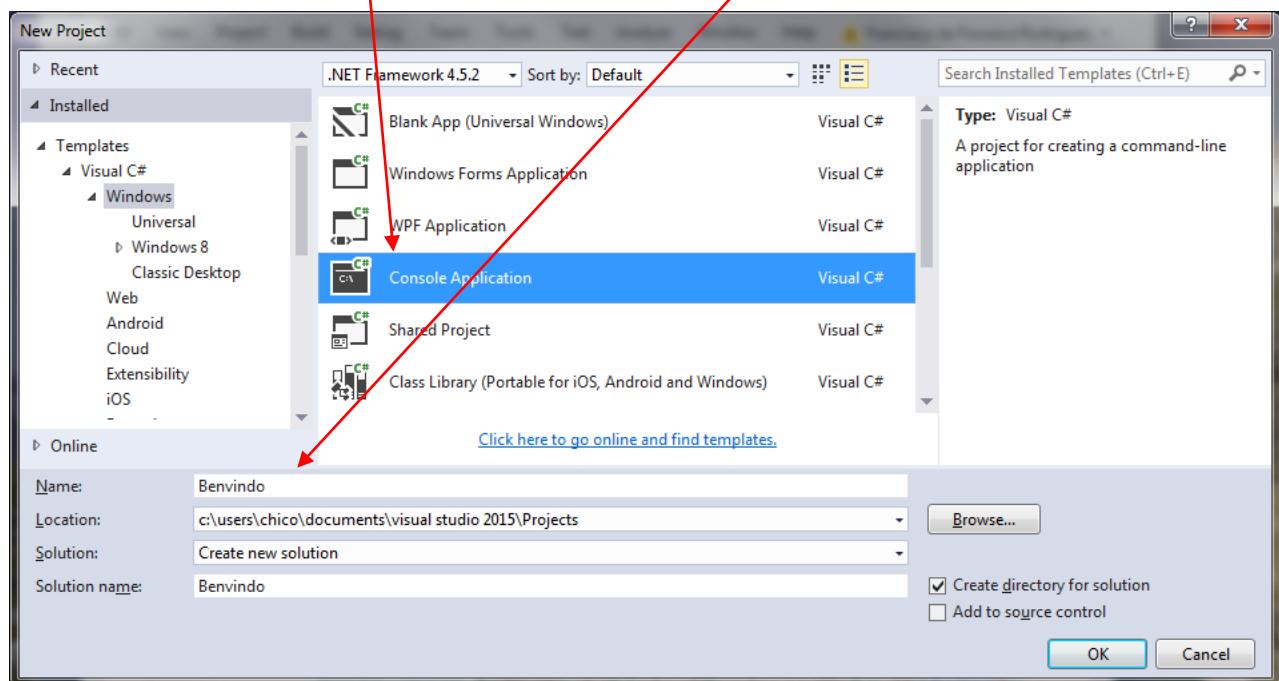
```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

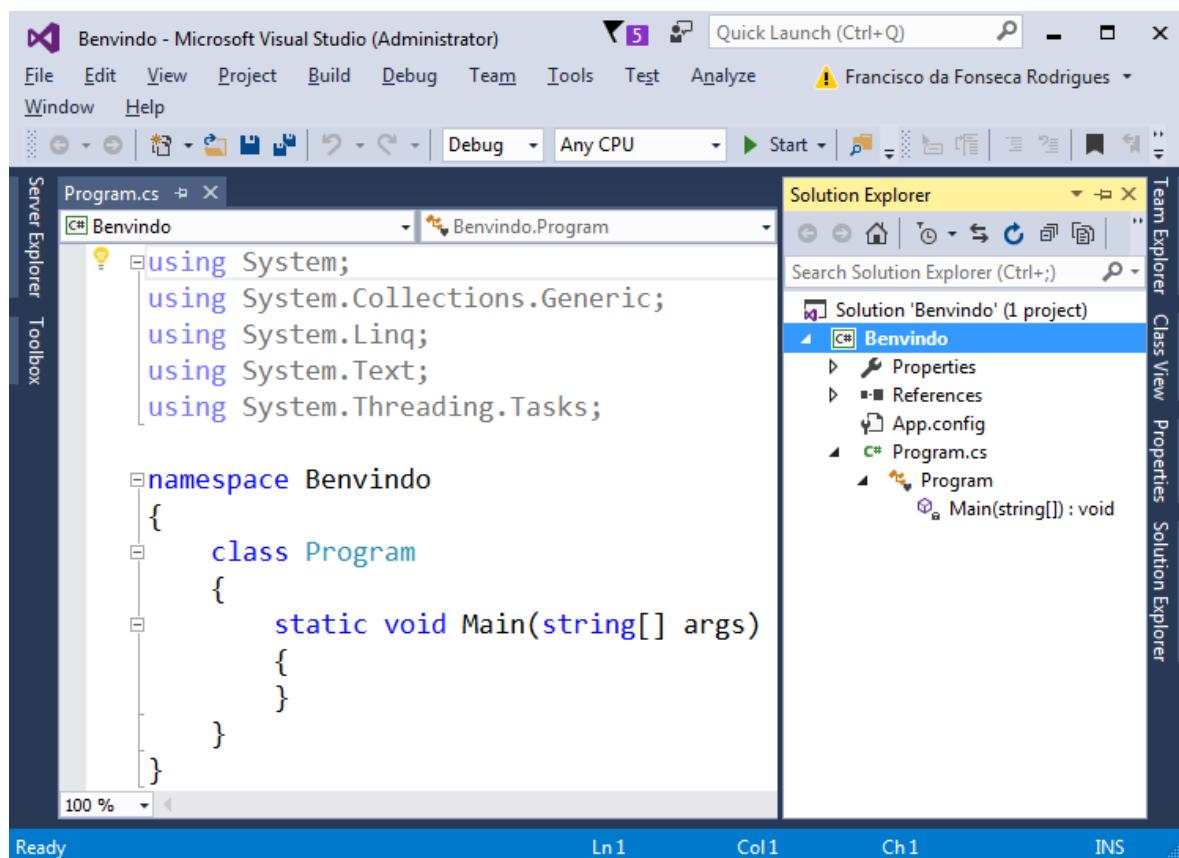
namespace apListaAlunos
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}

```

Vamos começar a aprender os comandos da linguagem através de aplicações console. Portanto, crie uma nova aplicação console, nomeando-a como **Benvindo**.



O resultado será semelhante ao visto abaixo:



Caso a janela do Solution Explorer não apareça, selecione a opção de menu View | Solution Explorer. Clique na figura do “percevejo” para travar essa janela, de modo que ela não deslize.

Program.cs* ▾ X

Benvindo Benvindo.Program Main(string[] args)

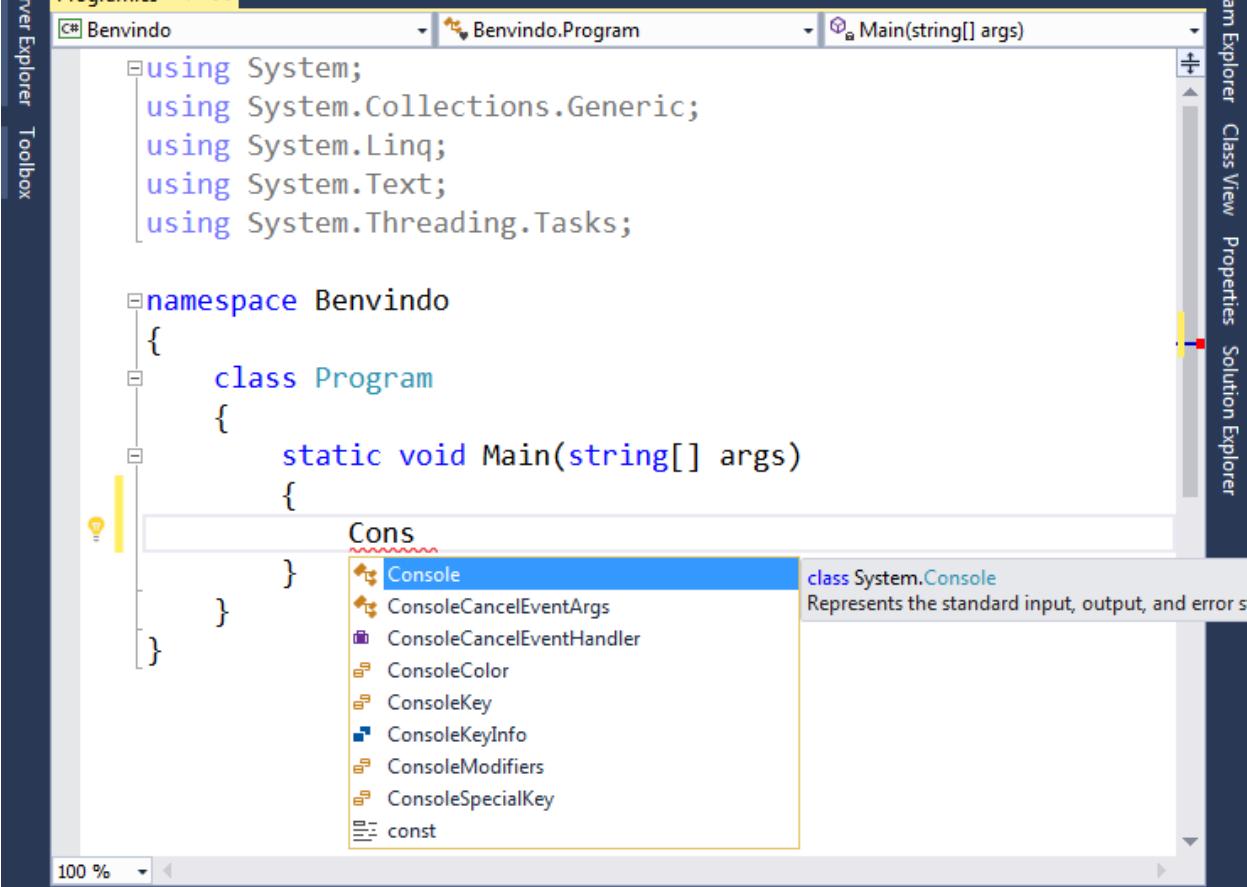
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Benvindo
{
    class Program
    {
        static void Main(string[] args)
        {
            Cons
        }
    }
}
```

100 %

277

Team Explorer Class View Properties Solution Explorer



Digite o texto “ **Console.WriteLine(“Benvindo à programação em C#”);** ” no corpo do método Main(), como vemos na figura abaixo:

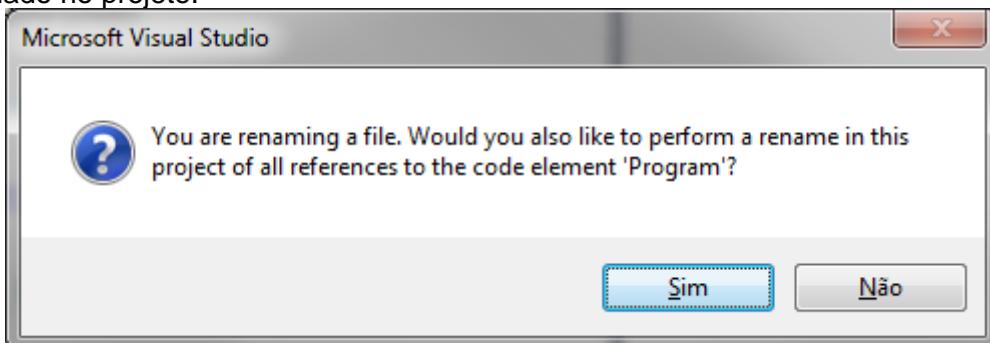
Observe o resultado do processo IntelliSense do Visual Studio: conforme você digita, o ambiente informa “dicas” e auxilia sua digitação.

Comentários em C# seguem as mesmas regras de Java e C++: // para comentários de linha, /* e */ para comentários em várias linhas do programa.

Using é a diretiva, gerada pelo ambiente de desenvolvimento, que declara que o programa usará os recursos do **namespace** System e os das linhas seguintes. Um **namespace** é como um package de Java, e agrupa vários recursos .Net em categorias relacionadas. Como o .Net Framework possui inúmeros namespaces prontos, contendo recursos que implementam várias classes, existem uma ampla gama de atividades que podem ser usadas por um programa, sem que o programador tenha que desenvolvê-las a partir do zero.

As classes pré-definidas no .Net Framework são conhecidas como o .Net Framework Class Library. Console é uma das classes descritas no namespace System.

Em seguida, vemos a declaração do namespace Benvindo, que representa nosso programa, e de uma classe chamada Program, subordinada a esse namespace. Se você desejar modificar o nome dessa classe, para pBenvindo, por exemplo, deve clicar com o botão direito no item Program.cs do Solution Explorer e selecionar a opção [Rename] no menu suspenso que aparecerá. Em seguida digite o novo nome do programa. Será exibida a janela abaixo, perguntando se você deseja alterar o nome Program em todas as localidades em que ele é referenciado no projeto:



Responda [Sim] e observe o resultado no código fonte de seu programa:

The screenshot shows the Microsoft Visual Studio interface. The code editor window displays the following C# code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Benvindo
{
    class pBenvindo
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Benvindo à programação em C#");
        }
    }
}
```

The Solution Explorer window on the right shows a project named 'Benvindo' containing files: 'Properties', 'References', 'App.config', and 'pBenvindo.cs'. The 'pBenvindo.cs' file is selected. A red arrow points from the class name 'pBenvindo' in the code editor to the file 'pBenvindo.cs' in the Solution Explorer.

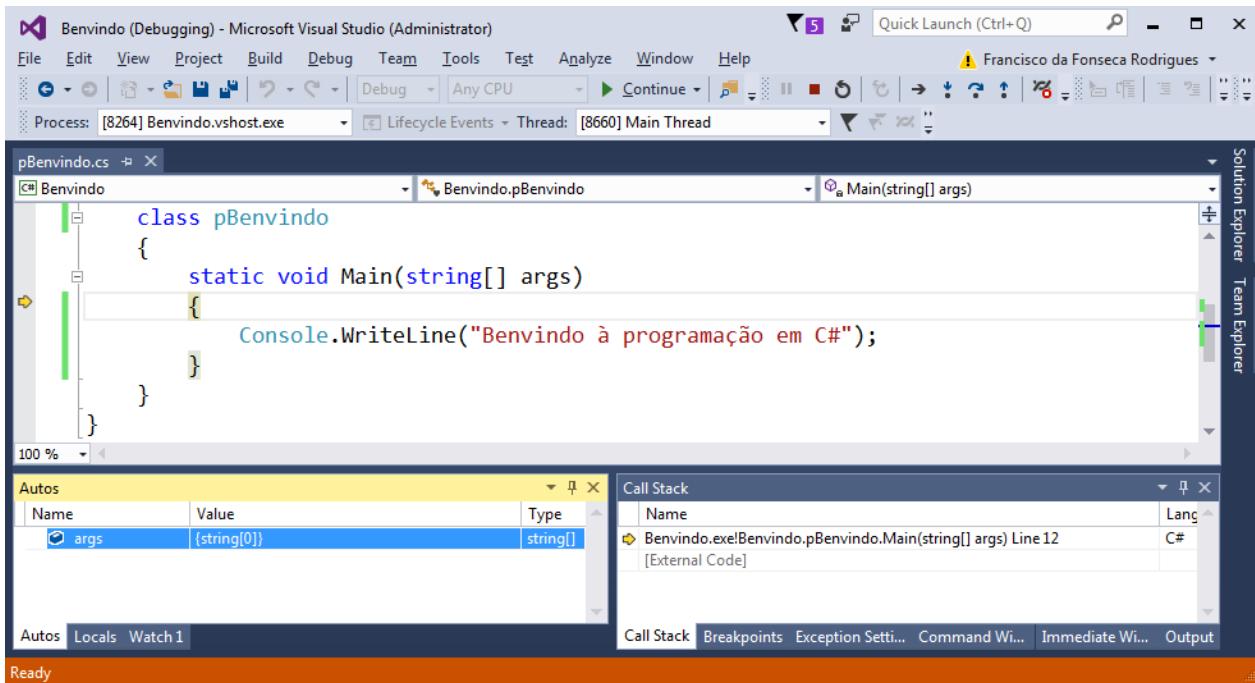
A classe **Console** permite que um programa exiba informação na saída padrão de um computador. Ela fornece métodos que permitem ao programa exibir strings e outros tipos de informação na linha de comandos do Windows. **WriteLine** é o método que executa a operação de exibição, colocando o cursor da tela na linha seguinte após a exibição.

Você poderia, também, escrever `System.Console.WriteLine`, pois `Console` é uma classe descrita no namespace `System`. No entanto, por ter escrito `Using System` no início do programa, isso passa a não ser necessário. Assim, economizamos tempo e deixamos o programa menos confuso.

A execução de um programa começa no método `Main()`.

Para compilar o programa, vá na opção de menu `Build | Build Solution`. Se não houver erros de sintaxe, será criado o arquivo `pBenvindo.exe`, com código MSIL. Para executá-lo, escolha a opção `Debug | Start Without Debugging`. Isso fará com que uma janela de linha de comandos do Windows seja exibida, o programa será executado e uma mensagem solicitando que você pressione qualquer tecla será também exibida.

Se você usar a opção `Debug | Start Debugging`, a janela aparecerá sem a mensagem final, de forma que a tela não ficará sendo exibida após o programa ter sido executado. A tecla `[F11]` permite a execução passo-a-passo:



Na string exibida por `Console.WriteLine` podemos informar caracteres de sequência de escape, que introduzem caracteres especiais no texto exibido, como vemos abaixo:

Sequência de Escape

\n
\t
\r
\\"
\”

Descrição

Newline. Posiciona o cursor na linha seguinte
Tabulação horizontal
CR, Enter. Posiciona o cursor no início da linha atual
Usado para exibir o caracter \ (barra)
Usado para exibir o caracter “ (aspas)

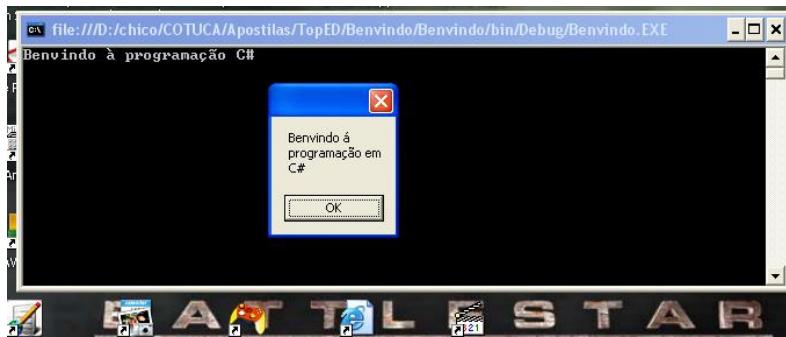
Podemos também usar uma caixa de mensagem para exibir os dados desejados. A classe `MessageBox` é agrupada no namespace `System.Windows.Forms`, que é usada, geralmente, em aplicações Windows Forms. Uma aplicação console como a que estamos desenvolvendo, por padrão não utiliza formulários windows, nem componentes gráficos. No entanto, é possível criar uma referência a esse namespace na aplicação console.

No Solution Explorer, clique em References, accese a opção Add Reference e a janela de adição de referências será exibida.

Na aba [.Net], procure o nome `System.Windows.Forms`, como vemos na próxima figura, e o selecione, pressionando o botão [OK]. Na lista de referências a namespaces do seu projeto aparecerá o namespace selecionado.

Agora, digite `using System.Windows.Forms;` no seu programa, e após ao comando que exibe a mensagem na linha de comando, digite:

```
MessageBox.Show("Benvindo à programação em\nC#");
```



Programa que soma dois valores:

Salve sua aplicação atual e crie uma nova aplicação console (File | New Project | Console Application) chamando-a Adicao, e digite o código abaixo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Adicao
{
    class Program
    {
        static void Main(string[] args)
        {
            string valor1, // primeiro valor digitado
                  valor2; // segundo valor digitado

            int numero1, // primeiro valor a somar
                numero2, // segundo valor a somar
                soma; // soma de numero1 e numero2

            // solicita ao usuário e lê os valores a somar
            Console.WriteLine("Digite o primeiro valor:");
            valor1 = Console.ReadLine();

            Console.WriteLine("Digite o segundo valor:");
            valor2 = Console.ReadLine();

            // converte as cadeias lidas do teclado
            // para valores inteiros
            numero1 = Int32.Parse(valor1);
            numero2 = Convert.ToInt32(valor2);

            // soma os números digitados
            soma = numero1 + numero2;

            // exibe o resultado
            Console.WriteLine("\nA soma é {0}\nObrigado!", soma);
        }
    }
}
```

Programa 2.1 – Adição de valores inteiros

Observe que várias dessas linhas já tinham sido criadas pelo ambiente de desenvolvimento.

O método ReadLine() da classe Console lê uma string do teclado e a armazena numa variável string. Note que, em C#, os tipos primitivos (chamados também de **tipos de valor**) são escritos com letras minúsculas, e que **string** é um desses tipos.

Observe também que há mais de uma maneira de converter a string lida para um valor inteiro. No .Net Framework existem classes que armazenam tipos, como **Int32**, **Int64**, **Double**, etc. A classe **Int32** é a base para o tipo primitivo **int** e pode ser usada para analisar uma string e converter seu conteúdo para um valor numérico inteiro. Para isso, usada-se o seu método Parse.

Por outro lado, existe a classe **Convert**, que contém métodos para efetuar conversões de qualquer tipo para outro, dentro das limitações das conversões. Assim, Convert.ToInt32 converte uma string para um número inteiro. Convert.ToDateTime converte uma string para uma data, e assim sucessivamente.

Console.WriteLine é um método com várias assinaturas, ou seja, é polimórfico. Uma de suas formas permite a exibição de vários valores de tipos diferentes, que podem, inclusive, serem formatados para apresentação na tela. No comando

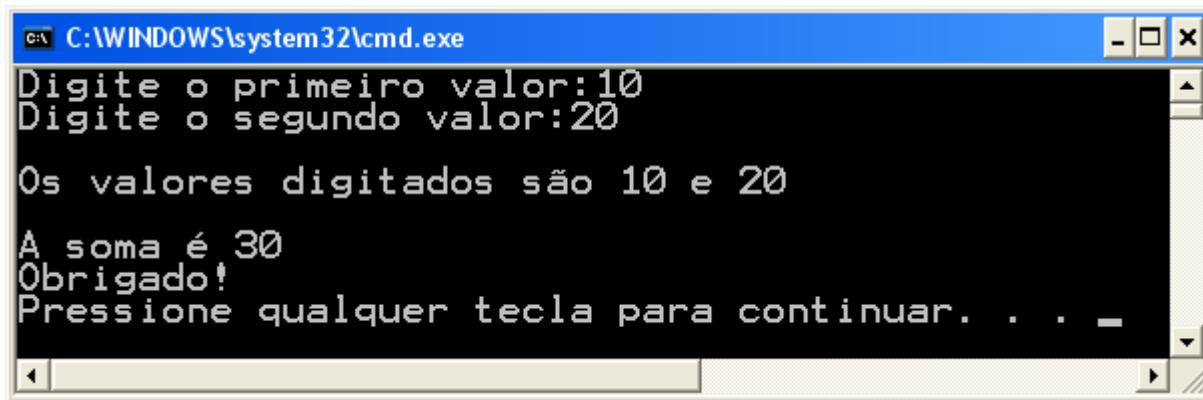
```
Console.WriteLine("\nA soma é {0}\nObrigado!", soma);
```

vemos que uma mensagem “A soma é “ será exibida. O indicador {0} Informa que em seguida a essa mensagem será escrito o primeiro valor (de índice 0) da lista de valores que seguem a mensagem. Como o primeiro valor seguinte à mensagem é a variável soma, seu valor será exibido, e após ele, uma linha será pulada (\n) e a frase “Obrigado!” será exibida. Após isso, o programa é terminado.

Se houver mais de um valor a ser exibido, basta incluirmos seus indicadores na cadeia que é o primeiro parâmetro do método como, por exemplo, vemos abaixo:

```
Console.WriteLine("\nOs valores digitados são {0} e {1}", numero1, numero2);
Console.WriteLine("\nA soma é {0}\nObrigado!", soma);
```

O resultado da execução desse programa vem abaixo:



```
Digite o primeiro valor:10
Digite o segundo valor:20
Os valores digitados são 10 e 20
A soma é 30
Obrigado!
Pressione qualquer tecla para continuar. . . -
```

Uma outra maneira de compor o comando de leitura dos valores numéricos poderia ser:

```
Console.Write("Digite o primeiro valor:");
numero1 = Int32.Parse(Console.ReadLine());
```

Os operadores aritméticos de C# são os mesmos de C, C++ e Java, com as mesmas precedências para cálculo dos resultados.

Os operadores relacionais, que fazem comparações entre valores, são também idênticos a C, C++ e Java.

O operador + é também usado para concatenar strings. Em C#, podemos concatenar strings com números, como vemos no comando abaixo:

```
Console.WriteLine("\nA soma é " + soma + "\nObrigado!");
```

Isso ocorre porque C# permite sobrepor operadores, e o operador + possui uma versão que permite somar valores numéricos e outra versão que permite concatenar valores de qualquer tipo com strings.

Palavras Reservadas de C#

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	int	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void	volatile	while

Comparação entre comandos de Java e C#

Os comandos **if** (com e sem **else**), **switch**, **while**, **do-while** e **for** são idênticos em C# e Java, tanto em sintaxe quanto em funcionamento.

Os operadores de atribuição **=**, **+=**, **-=**, ***=**, **/=** e **%=** funcionam da mesma forma que em Java.

Os operadores de inscemento **(++)** e decremento **(--)** também funcionam exatamente da mesma maneira.

O próximo programa (programa 2.2) deve ser criado como uma nova aplicação console, com nome Somalnteiros. Após adicionar a referência ao namespace **System.Windows.Forms**, digite o código abaixo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace SomaInteiros
{
    class Program
    {
        static void Main(string[] args)
        {
            int soma = 0;

            for (int numero = 2; numero <= 100; numero += 2)
                soma += numero;

            MessageBox.Show("A soma é " + soma,
                           "Soma dos inteiros pares de 2 a 100",
                           MessageBoxButtons.OK,
                           MessageBoxIcon.Information);
        }
    }
}
```

Programa 2.2 – Somatória dos inteiros pares de 2 a 100

Observe os argumentos do método **Show** do **MessageBox**. O primeiro argumento é a mensagem exibida, o segundo é o título da janela de mensagem, o terceiro indica os botões que serão exibidos e o último informa qual ícone será exibido na janela, como vemos no resultado abaixo:



As tabelas a seguir descrevem os ícones e botões disponíveis para uso em MessageBox. Esses recursos estão disponíveis nos objetos MessageBoxButtons e MessageBoxIcon, respectivamente.

Ícones do MessageBox

MessageBoxIcon.Exclamation



Descrição

Geralmente usado para advertir o usuário sobre problemas potenciais.

MessageBoxIcon.Information



Especifica que a caixa de mensagem possui uma informação para o usuário.

MessageBoxIcon.Question



Geralmente usado em mensagens que perguntam algo ao usuário.

MessageBoxIcon.Error



Alerta o usuário sobre erros ou mensagens importantes.

Botões do MessageBox

MessageBoxButton.OK

Descrição

Especifica que a caixa de mensagem deverá incluir um botão **OK**.

MessageBoxButton.OKCancel

Especifica que a caixa de mensagem deverá incluir os botões **OK** e **Cancel**. Adverte o usuário sobre alguma condição e lhe permite continuar ou cancelar uma operação.

MessageBoxButton.YesNo

Especifica que a caixa de mensagem deverá conter os botões **Yes** e **No**. Usado para fazer perguntas ao usuário.

MessageBoxButton.YesNoCancel

Especifica que a caixa de mensagem deverá incluir os botões **Yes**, **No** e **Cancel**.

Geralmente usado para fazer perguntas ao usuário mas, também, permitindo que o usuário cancele a operação.

MessageBoxButton.RetryCancel

Especifica que a caixa de mensagem deverá incluir os botões **Retry** e **Cancel**.

Geralmente usado para informar o usuário sobre uma operação falha e permitir-lhe refazer ou cancelar essa operação.

MessageBoxButton.AbortRetryIgnore

Especifica que a caixa de mensagem deverá incluir os botões **Abort**, **Retry** e **Ignore**.

Geralmente usada para informar o usuário que alguma de uma série de operações falhou, permitindo-lhe abortar a série de operações, refazer a operação que falhou ou ignorar a falha e continuar o processo.

O próximo exemplo (programa 2.3) usa um comando **for** para calcular juros compostos. Considere o seguinte problema:

Uma pessoa investe R\$1000.00 em uma aplicação que rende juros de 5% ao ano. Assumindo que todos os juros são deixados no depósito, calcule e exiba a quantia de dinheiro na aplicação ao final de cada ano, pelos próximos 10 anos. Para determinar essas quantias, use a fórmula:

$$a = p(1 + r)^n$$

onde

p é a quantia original investida (ou seja, o principal)

r é a taxa anual de juros

n é o número de anos

a é a quantia depositada ao final do n-ésimo ano

Na solução usaremos variáveis do tipo primitivo **decimal**, que é usado para cálculos monetários. Valores reais, em C#, são tratados implicitamente como **double**, da mesma forma que valores inteiros são tratados como sendo do tipo **int**. Para usar os valores reais do programa abaixo como **decimal**, teremos de realizar uma conversão explícita de tipos, ou **type cast**. Também é possível especificar que uma constante é do tipo **decimal** colocando uma letra **m** após o valor, como em 1000.0m.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace Juros
{
    class Program
    {
        static void Main(string[] args)
        {
            decimal quantia, principal = (decimal)1000.00;
            double taxa = 0.05;
            string resultado = "Ano\tQuantidade depositada\n";

            for (int ano = 1; ano <= 10; ano++)
            {
                quantia = principal * (decimal)Math.Pow(1.0 + taxa, ano);

                resultado += ano + String.Format("\t{0:C}\n", quantia);
            }
            MessageBox.Show(resultado, "Juros compostos",
                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }
}
```

Programa 2.3 – Cálculo de juros compostos

Não se esqueça de colocar os caracteres \t e \n onde indicados na string resultado.

Ao ser executado, esse programa resulta na janela de mensagem abaixo:



Note que usamos o método estático Pow da classe Math. Como C# não possui um operador de exponenciação, podemos usar Math.Pow para de cálculos dessa natureza. O primeiro argumento é a base e o segundo argumento é o expoente. Portanto, no comando acima calculamos $(1+taxa)^{ano}$.

Dentro do for, concatenamos cada linha a ser exibida dentro da string resultado, para ao final do for ela ser exibida com o MessageBox.Show.

Usamos a chamada ao método Format da classe String, que converte a variável quantia para uma string de forma que ela exiba duas casas decimais. String.Format utiliza os códigos de cadeias de formatação para representar valores numéricos e monetários de acordo com a forma apropriada ao ambiente de execução. Por exemplo, no Brasil, valores monetários são formatados com duas casas decimais, separadas da parte inteira por vírgula, e com pontos separando cada milhar. Já nos Estados Unidos seria o contrário. Note também que o símbolo monetário (R\$ no caso acima) foi usado em acordo com o ambiente de execução.

O primeiro argumento de String.Format é a cadeia de formatação. Já vimos essas cadeias em comandos anteriores, na forma de {0} e {1}, para indicar valores que seriam escritos. Nesses casos, os valores numéricos 0 e 1 indicam a seqüência das informações que serão escritas, associando-as com variáveis e expressões. No caso de cadeias de formatação, tais como "{0:C}", o dígito 0 ainda representa o argumento a ser exibido e sua posição na lista de argumentos. A informação especificada após : representa a formatação desejada ao valor exibido, e geralmente é chamada de código de formatação. A letra C indica que o valor será exibido como um valor monetário (**currency**).

Há vários outros códigos de formatação, cujas descrições podem ser obtidas na documentação do MSDN. Abaixo temos algumas descrições que poderão ser úteis:

Código de Formatação**C ou c**

Formata a cadeia como valor monetário. O valor é precedido com o símbolo monetário configurado no ambiente de execução. Dígitos são separados com o caracter separador decimal configurado e estabelece o número de casas decimais em duas, por default.

D ou d

Formata a cadeia com um valor decimal. Exibe números como inteiros.

N ou n

Formata a cadeia com vírgulas e duas casas decimais.

E ou e

Formata a cadeia em notação científica - default de 6 casas decimais.

F ou f

Formata a cadeia com um número fixo de casas decimais (duas, por default).

X ou x

Formata a cadeia como hexadecimal.

Funções embutidas importantes

As funções abaixo são métodos do namespace Math, e podem ser chamadas com Math.função ou, caso se tenha escrito “using Math;” no inicio do programa, pode-se simplesmente invocar o nome da função desejada, com seus parâmetros.

Função	Descrição	Exemplo
Abs(x)	valor absoluto (modulo) de x	Abs(23.7) é 23.7 Abs(0) é 0 Abs(-23.7) é 23.7
Ceiling(x)	arredonda x para o menor inteiro não menor que x	Ceiling(9.2) é 10.0 Ceiling(-9.8) é -9.0
Cos(x)	cosseno trigonométrico de x (x em radianos)	Cos(0.0) é 1.0
Exp(x)	exponencial e^x	Exp(1.0) aproximadamente 2.71828182845 Exp(2.0) aproximadamente 7.38905609893
Floor(x)	arredonda x para o maior inteiro não maior que x	Floor(9.2) é 9.0 Floor(-9.8) é -10.0
Log(x)	logaritmo natural de x (base e)	Log(2.7182818284590451) é 1.0 Log(7.3890560989306504) é 2.0
Max(x, y)	o maior valor dentre x e y	Max(2.3, 12.7) é 12.7 Max(-2.3, -12.7) é -2.3
Min(x, y)	o menor valor dentre x e y	Min(2.3, 12.7) é 2.3 Min(-2.3, -12.7) é -12.7
Pow(x, y)	x elevado ao expoente y (x^y)	Pow(2.0, 7.0) é 128.0 Pow(9.0, .5) é 3.0
Sin(x)	seno trigonométrico de x (x em radianos)	Sin(0.0) é 0.0
Sqrt(x)	raiz quadrada de x	Sqrt(900.0) é 30.0 Sqrt(9.0) é 3.0
Tan(x)	tangente trigonométrica de x (x em radianos)	Tan(0.0) é 0.0

Constantes importantes do namespace Math

Math.PI = 3.14159265358979323846

Math.E = 2.7182818284590452354

3. Namespaces e Métodos

Como vimos, C# contém muitas classes predefinidas que são agrupadas em namespaces.

Este código pré-existente é conhecido, coletivamente, como Framework Class Library (FCL). O código real para as classes é localizado em arquivos .dll chamados **assemblies**.

Como já vimos, em C# as diretivas comandos **using** especificam **os namespaces** que usamos em cada programa.

Por exemplo, um programa inclui o comando **using System**; para informar o compilador que estamos usando o namespace **System**. Este comando **using** nos permite escrever **Console.WriteLine** ao invés de **System.Console.WriteLine** em todo o programa. Para usar uma classe de um namespace específico, devemos adicionar uma referência ao assembly apropriado, como fizemos para usar o MessageBox em um programa console, na página 188. Referências ao assembly do namespace **System** são adicionadas automaticamente — outros assemblies devem ser adicionados explicitamente.

Abaixo temos alguns dos namespaces mais usados, presentes na FCL, e sua descrição.

Namespace	Descrição
System	Contém classes essenciais e tipos de dados, tais como int , double , char , etc. É referenciado implicitamente por todos os programas C#.
System.Data	Contém classes que formam o ADO.NET, usado para acesso e manipulação de bancos de dados.
System.Drawing	Contém classes usadas para desenhos e gráficos.
System.IO	Contém classes usadas para entrada e saída de dados, tais como com arquivos.
System.Threading	Contém classes para multithreading, usadas para executar várias partes de um programa simultaneamente.
System.Windows.Forms	Contém classes usadas para criar interfaces gráficas com o usuário.
System.Xml	Contém classes para processar dados em XML.

O conjunto de namespaces disponíveis na FCL é muito grande. Além dos namespaces descritos na tabela acima, a FCL inclui namespaces para gráficos complexos, interfaces gráficas com o usuário avançadas, impressão, rede avançada, segurança, multimídia, acessibilidade (para pessoas com necessidades especiais) e muito mais. Para conhecer os vários namespaces da FCL, consulte “.Net Framework class library” na documentação do help ou na MSDN.

Variáveis

Há dois tipos de variáveis em C# - *variáveis locais* e *campos*. Uma variável local é uma variável declarada dentro de um método. Um campo é uma variável declarada dentro de uma *struct* (estrutura como um registro em Pascal) ou *class* (uma classe, semelhante às de Java e Delphi). O tipo de uma variável em C# pode ser classificado como "tipo de valor" ou "tipo de referência".

Tipos de Valor ou Value Type

Os tipos de valor de C# são os tipos simples, primitivos, e structs. Os tipos primitivos são **bool**, **char**, **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double** e **decimal**. A especificação de tipos da linguagem C# (**Common Type System**) define o intervalo de valores para cada tipo primitivo e o conjunto de operações suportadas por cada um deles.

Cada variável com tipo de valor é uma instância distinta do tipo usado para declará-la. Portanto, um comando de atribuição tal como o abaixo

```
y = x;
```

obtém o *valor* da variável x e o copia para dentro da variável y. Após a atribuição, x e y continuam como instâncias distintas que, nesse momento, possuem valores iguais.

Uma comparação da forma

```
if (x == y)  
{ /* ... */ }
```

verifica se os valores contidos nas variáveis x e y são iguais.

Tipos de Referência

Em C#, cada variável que não é declarada com um dos tipos por valor é de um *tipo de referência*. Esse tipo de variável pode ser imaginado como uma referência (ou um ponteiro) para um objeto de um outro tipo.

Cada objeto para o qual uma variável de referência aponta é uma instância de alguma *classe C#*. Em C#, instâncias de classes devem ser criadas explicitamente. Uma instância de uma classe é criada usando o operador new como abaixo:

```
Foo f = new Foo();
```

Se este comando de atribuição for seguido por outro comando de atribuição tal como

```
Foo g = f;
```

então tanto f quanto g se referirão ao mesmo objeto! Note que isto é diferente do que acontece quando você atribui uma variável com tipo de valor a outra.

Uma comparação da forma

```
if (f == g)  
{ /* ... */ }
```

geralmente verificará se f e g se referem à mesma instância de objeto (desde que o operador de igualdade não tenha sido sobreposto). Se f e g se referem a instâncias de objeto distintas, mesmo que o conteúdo desses objetos seja igual, o teste resulta em falso. Para testar a igualdade do conteúdo de duas instâncias distintas de um objeto da mesma classe, precisamos usar o método Equals, como abaixo:

```
if (f.Equals(g))  
{ /* ... */ }
```

Referências Nulas

Em C#, é possível, para uma variável de tipo de referência não se referir a nada. Uma referência que não se refira a nada é chamada de *referência nula*. Por default, uma referência não iniciada vale **null**.

Podemos atribuir uma referência nula a uma variável explicitamente, como abaixo:

```
f = null;
```

Também, podemos testar explicitamente se uma variável é uma referência nula, como abaixo:

```
if (f == null)
{ /* ... */ }
```

Mecanismos de Passagem de Parâmetros

Os mecanismos de passar parâmetros definem as maneiras pelas quais parâmetros são transferidos entre métodos quando um método chama outro. C# fornece dois mecanismos de passagem de parâmetros: passagem por valor e passagem por referência.

Passagem por Valor

Considere o par de métodos C# definidos no programa abaixo (3.1). Na linha 7, o método One chama o método Two. Em geral, cada chamada de um método inclui uma lista de argumentos (que pode ser vazia). Os argumentos especificados em uma chamada de método são chamados de parâmetros reais. Neste caso, há apenas um parâmetro real – y.

```
1. public class Exemplo
2. {
3.     public static void One()
4.     {
5.         int x = 1;
6.         Console.WriteLine(x);
7.         Two(x); // x é chamado de argumento ou parâmetro real
8.         Console.WriteLine(x);
9.     }
10.    public static void Two(int y) // y é chamado de parâmetro formal
11.    {
12.        y = 2;
13.        Console.WriteLine(y);
14.    }
15.    static void Main(string[] args)
16.    {
17.        One();
18.    }
19. }
```

Programa 3.1 – Passagem de parâmetros por valor

Na linha 10 o método Two é definido como aceitando um único argumento de tipo int, chamado y. Os argumentos que aparecem na definição de um método são chamados de parâmetros formais. Neste caso, o parâmetro formal é um *tipo de valor*.

A semântica de uma passagem por valor funciona da seguinte maneira: o efeito da definição de um parâmetro formal é criar uma variável local do tipo especificado no método dado. Por exemplo, o método Two tem uma variável local de tipo int chamada y. Quando o método é chamado, os **valores** dos *parâmetros reais* são usados para atribuir os valores dos *parâmetros formais* antes que o corpo do método seja executado.

Já que os parâmetros formais forçam a criação de variáveis locais, se um novo valor é atribuído a um parâmetro formal, este valor não tem efeito algum sobre o parâmetro real. Portanto, a saída produzida pela execução do método One definido acima é:

Passagem por Referência

Considere os métodos One e Two definidos no programa abaixo (3.2). A única diferença entre este código e o código dado no programa anterior é o uso da palavra reservada **ref** na **definição** e **na chamada** do método Two.

Neste caso, o parâmetro formal *y* é declarado como sendo a **referência** a um **int**. Portanto, quando o método é chamado, o parâmetro real deve também ser uma referência a um int. A expressão *ref x* na linha 7 fornece uma referência à variável *x*. Assim sendo, neste caso a passagem de parâmetro é feita usando o mecanismo de *passagem por referência*.

```
1. public class Exemplo
2. {
3.     public static void One()
4.     {
5.         int x = 1;
6.         Console.WriteLine(x);
7.         Two(ref x);
8.         Console.WriteLine(x);
9.     }
10.    public static void Two(ref int y)
11.    {
12.        y = 2;
13.        Console.WriteLine(y);
14.    }
15.    static void Main(string[] args)
16.    {
17.        One();
18.    }
19. }
```

Programa 3.2 – Passagem de parâmetros por referência

Um parâmetro formal passado por referência não é uma variável. Quando um método que possui um parâmetro formal por referência é chamado, o efeito da chamada é associar a referência ao parâmetro real correspondente. Dessa forma, a referência se torna um nome alternativo para o parâmetro real correspondente.

Um parâmetro formal por referência pode ser usado dentro do método chamado em todo lugar onde uma variável poderia ser usada. Em particular, se o parâmetro formal por referência é usado onde seu valor é necessário, é o valor do parâmetro real que é obtido. Da mesma maneira, se o parâmetro por referência é usado onde uma referência é necessária, é a referência ao parâmetro real que é obtida. Portanto, a saída obtida pelo método One definido no programa anterior é:

1
2
2

Diferenças entre os mecanismos

Claramente, a abordagem usada para passagem de parâmetros coloca restrições na funcionalidade do método chamado: quando se usa a passagem por valor, o método chamado não pode modificar os parâmetros reais; quando se usa a passagem por referência, o método é capaz de modificar os valores dos parâmetros reais. Além disso, as duas abordagens apresentam requisitos distintos de tempo e espaço em memória que precisam ser compreendidos a fim de se fazer a seleção correta do mecanismo usado.

Passagem por valor cria uma variável local e a inicia com o valor do parâmetro real. Isto significa espaço usado (na pilha) para a variável local e que é dispendido tempo para iniciá-la. Para variáveis simples, pequenas, essas penalidades são insignificantes. No entanto, se a variável é uma estrutura grande, as penalidades de tempo e de espaço podem se tornar proibitivas.

Por outro lado, passagem por referência não cria uma variável local nem requer a cópia dos valores presentes nos parâmetros reais. No entanto, devido à forma como esse parâmetro é implementado, a cada vez que se usa um parâmetro formal passado por referência para se acessar o parâmetro real, um pequeno gasto extra de tempo ocorre, para se acessar o local de memória cujo endereço é indicado pelo parâmetro por referência (esta operação se chama de referenciar). Como resultado, tipicamente é mais eficiente passar-se pequenas variáveis por valor e variáveis grandes (como um vetor), por referência.

Parâmetros Out

C# realmente suporta dois diferentes “ sabores ” de passagem por referência : ref e out . Out é um casos especiais de ref .

Nas primeiras versões de C# havia uma passagem de parâmetro “in”: o parâmetro é passado por referência para “entrada” em um método. Especificamente, o endereço do parâmetro real é associado ao parâmetro formal, de forma que o acesso continua sendo indireto, mas não é possível alterar o valor do parâmetro no corpo do método. Assim, o acesso continua sendo feito pelo endereço, mas impede-se que novos valores sejam atribuídos ao parâmetro real. Atualmente não existe mais esse tipo de passagem; usa-se ref em seu lugar, tomando-se o cuidado de não alterar o valor do parâmetro quando isso não é desejado pela lógica do programa.

Já um parâmetro out é passado por referência mas apenas o valor de saída pode ser acessado, fora do método. Especificamente, o parâmetro formal recebe um valor, antes do retorno do método ao local de chamada, mas não pode usar o parâmetro internamente, em qualquer operação que acesse o valor do parâmetro formal, antes que um valor seja atribuído a ele dentro do método. O último valor atribuído será o valor do parâmetro real, após retornar-se do método.

Passando Tipos de Referência

O programa abaixo ilustra a passagem de parâmetros de tipos de referência, ou seja, de objetos (ou variável que contêm o endereço de outra variável). Neste caso, as variáveis x , y e z são do tipo Obj , que é um tipo de referência. Assim, x , y e z se referem a instâncias da classe Obj , definida nas linhas 5 a 8:

```
1. namespace ReferenceTypeParam
2. {
3.     public class Exemplo
4.     {
5.         public class Obj
6.         {
7.             public int campo = 1;
8.         }
9.
10.        public static void One()
11.        {
12.            Obj x = new Obj();
13.            Console.WriteLine(x.campo);
14.            Two(x);
15.            Console.WriteLine(x.campo);
16.            Three(ref x);
17.            Console.WriteLine(x.campo);
18.        }
19.    }
20. }
```

```
17.     }
18.     public static void Two(Obj y)
19.     {
20.         y.campo = 2;
21.         Console.WriteLine(y.campo);
22.     }
23.     public static void Three(ref Obj z)
24.     {
25.         z = new Obj();
26.         Console.WriteLine(z.campo);
27.     }
28. }
29. }
```

Programa 3.3 – Passando objetos por referência

A semântica da passagem de parâmetros para tipos de referência funciona exatamente como aquela para tipos de valor: na passagem por valor, o efeito da definição do parâmetro formal é criar uma variável local do tipo especificado pelo método. Por exemplo, o método Two tem uma variável local de tipo Obj chamada y. Quando o método é chamado, os parâmetros reais são atribuídos para os parâmetros formais antes que o corpo do método comece a ser executado. Já que x e y são tipos de referência, quando atribuímos y a x, fazemos com que ambos se refiram à mesma instância da classe Obj. Portanto, o método modifica a instância original de Obj.

Na passagem por referência, o parâmetro formal acaba sendo uma referência ao tipo de referência representado pelo parâmetro real. Por exemplo, no método Three o parâmetro formal z se refere ao parâmetro por referência x. assim, quando criamos uma nova instância da classe Obj e a atribuímos a z, a variável referenciada x é modificada, e x passa a apontar para a mesma instância criada que z aponta. Quando o fluxo de execução retorna do método Three, z é descartada (pois é uma variável local) e x continua apontando (referenciando) aquela instância criada dentro de Three.

Dessa forma, passamos uma referência como um parâmetro por referência.

A saída obtida pelo método One definido no programa acima é:

```
1
2
2
1
1
```

4. Objetos e Classes

Objetos e Classes

Uma **classe** C# define uma estrutura de dados que contém campos, métodos e tipos aninhados. Cada tipo em C# possui uma classe direta ou indiretamente derivada da classe **object**. A classe de um objeto determina o que ele é e como ele pode ser manipulado. Uma classe encapsula dados, operações e semânticas. Este encapsulamento é como um *contrato* entre o implementador da classe e o usuário dessa classe.

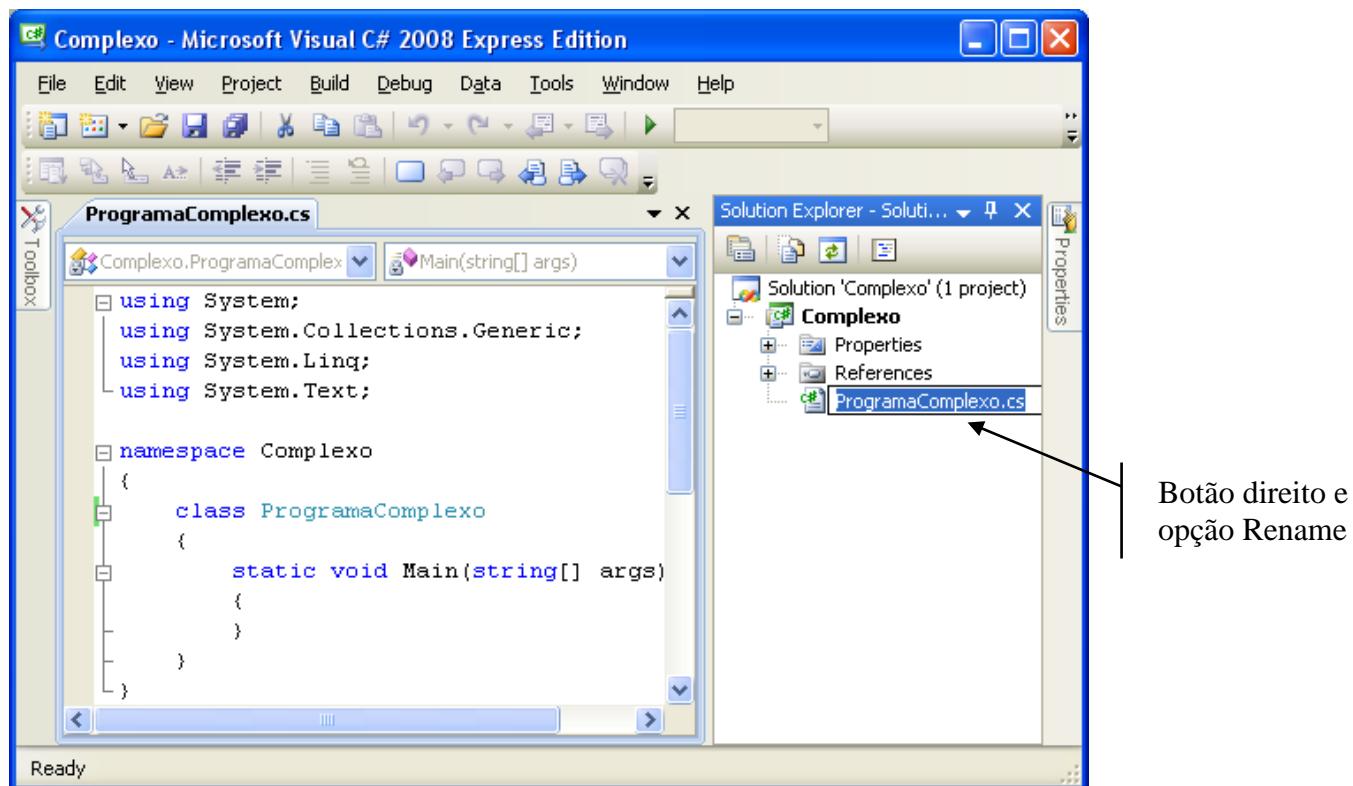
O construtor **class** é o que torna C# uma linguagem orientada a objetos. Uma definição de classe C# agrupa um conjunto de valores com um conjunto de operações. Classes facilitam a modularização e o ocultamento de informações. O usuário de uma classe manipula instâncias de objetos daquela classe somente através dos métodos fornecidos pela classe.

É comum que diferentes classes possuam características comuns. Classes distintas podem compartilhar valores comuns; elas podem realizar as mesmas operações; elas podem suportar interfaces comuns. Em C# tais relacionamentos são expressos usando os mecanismos de derivação e herança.

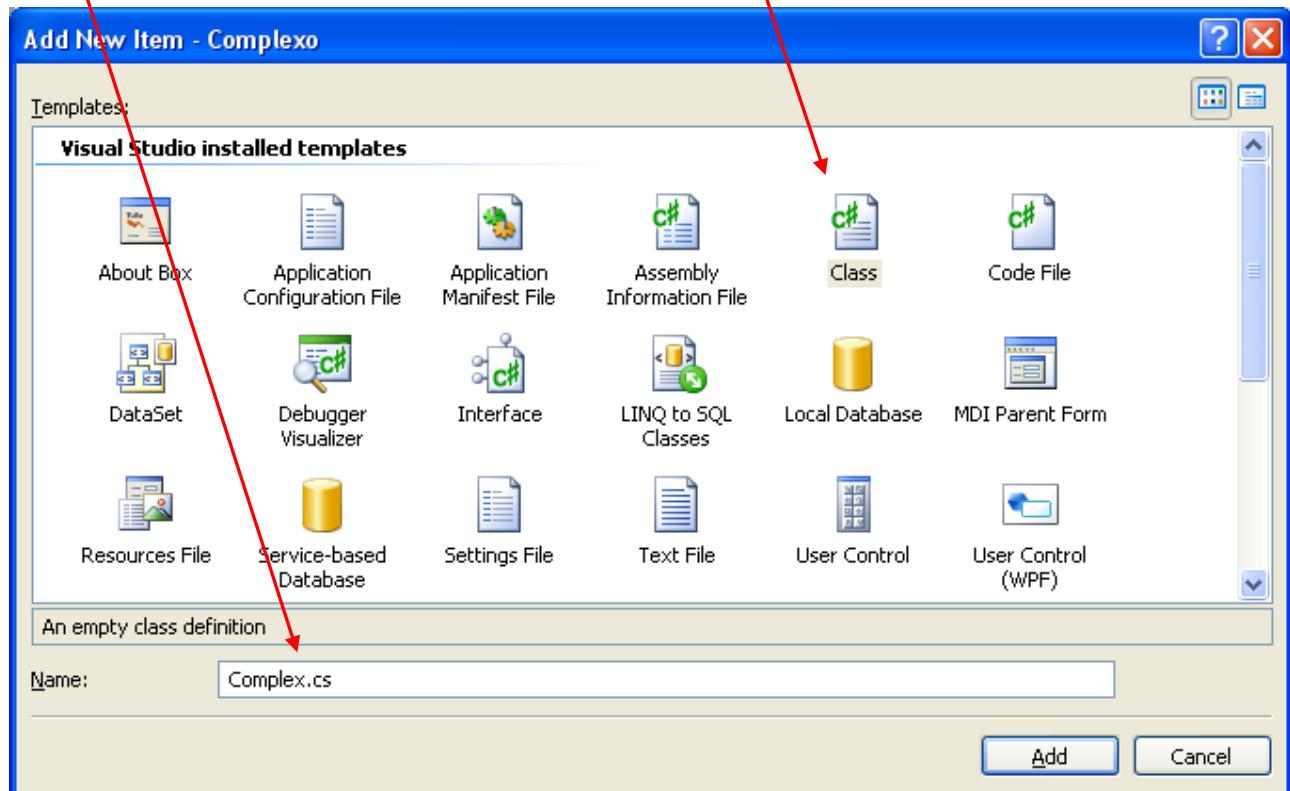
Membros de Classes : Campos e Métodos

Uma classe agrupa um conjunto de valores e um conjunto de operações. Os valores e as operações de uma classe são conhecidas como seus *membros*. *Campos* implementam os valores e *métodos* implementam as operações.

Crie um novo projeto, como uma aplicação console chamada Complexo. Renomeie o item Program.cs para ProgramaComplexo.cs, como vemos na figura abaixo:



Agora, adicione um novo item, que será uma classe. Para isso, pode-se usar o menu Projeto | Add New Item | Class ou clique no Solution Explorer com o botão direito e selecione a opção Add | New Item. Na janela que aparecerá, escolha o item Class e mude seu nome de Class1.cs para Complex.cs, como vemos na próxima figura:



Suponha que desejamos definir uma classe para representar números complexos. A definição da classe Complex mostrada abaixo ilustra como isto pode ser feito.

Abra a aba Complex.cs do Visual C# 2008 Express Edition e digite o código abaixo, após os comandos using:

```
namespace Complexo
{
    class Complex
    {
        private double real;
        private double imag;

        public double Real
        {
            get { return real; }
            set { real = value; }
        }

        public double Imag
        {
            get { return imag; }
            set { imag = value; }
        }
        // ...
    }
}
```

Dois campos, real e imag, são declarados. Eles representam as partes real e imaginária de um número complexo, respectivamente.

x = cmp.Real; cmp.Real = 2.5;

Também temos duas **propriedades**, Real e Imag, cada uma delas fornecendo acesso para obter (get) e atribuir (set) valores para os campos. Essas propriedades são chamadas de **accessors** (acessadores), pois são usadas para acessar as partes real e imaginária do número complexo.

Programa 4.1 – Classe Complex, seus campos e propriedades iniciais

Em C#, quando uma propriedade recebe um valor, automaticamente é chamado a parte set de sua declaração. O valor atribuído é associado à “variável” value e atribuído ao campo. Portanto, se em um programa você usar a classe Complex e atribuir um valor à propriedade Real (com maiúsculo), o campo real (com minúsculo) receberá esse valor, como vemos abaixo:

```
Complex numero = new Complex();  
numero.Real = 3.5; → chama o acessador set da propriedade Real e associa 3.5 a “value”,  
que é, então, atribuído para o campo real.
```

Cada instância de objeto da classe Complex contém seus dois campos. Considere as declarações de variáveis abaixo:

```
Complex c = new Complex();  
Complex d = new Complex();
```

Tanto c quanto d se referem a diferentes instâncias da classe Complex. Portanto, cada uma delas possui seu espaço de memória próprio e seus próprios campos real e imag. Os campos de um objeto são acessados usando o operador ponto. Por exemplo, c.real se refere ao campo real de c e d.imag se refere ao campo imag de d.

O programa abaixo também define as propriedades Real e Imag. Em geral, uma propriedade é um atributo de uma instância da classe. Novamente, o operador ponto é usado para especificar a propriedade sobre a qual uma operação é realizada. Por exemplo, c.Real = 1.0 invoca o acessador set da propriedade Real em c e Console.WriteLine(d.Imag) invoca o acessador get da propriedade Imag em d.

Construtores

Um *construtor* é um método que tem o mesmo nome que a classe, e que não tem um valor retornado. Três construtores foram definidos no programa abaixo. O propósito de um construtor é iniciar um objeto. Um construtor é invocado sempre que uma nova instância de uma classe é criada, ao se usar o operador new.

```
1. public class Complex  
2. {  
3.     private double real;  
4.     private double imag;  
5.  
6.     public Complex(double x, double y)  
7.     {  
8.         real = x;  
9.         imag = y;  
10.    }  
11.  
12.    public Complex()  
13.        : this(0, 0)  
14.    { }  
15.  
// ...
```

Programa 4.2 – Construtores da classe Complex

Considere a seguinte seqüência de declarações de variáveis:

```
Complex c = new Complex(); // chama Complex ()
```

```
Complex d = new Complex(2.0); // chama Complex (double)
Complex i = new Complex(0, 1); // chama Complex (double, double)
```

Considere o construtor com dois argumentos double, x and y (linhas 5 a 9). Este construtor inicia o número complexo atribuindo x e y para os campos real e imag, respectivamente.

Construtor sem argumentos

Como vemos no programa, pode-se criar construtores sem argumentos; por exemplo, o construtor sem argumentos é chamado quando uma variável é declarada e instanciada como abaixo:

```
Complex c = new Complex();
```

Porém, se não há construtores definidos em uma classe C#, o compilador fornece um construtor *default sem argumentos*. Esse construtor default não faz nada. Os campos simplesmente mantêm os seus valores iniciais.

O programa acima fornece uma implementação para o construtor sem argumentos da classe Complex (linhas 10-12). Esse construtor usa o inicializador chamado **this**. Em C# um construtor pode chamar outro construtor através do inicializador **this**. Neste caso, o construtor sem argumentos chama o construtor com dois argumentos, para definir como zero o valor tanto do campo real quanto do campo imag.

Propriedades e Acessadores

Uma propriedade em C# define um ou dois métodos para acessar um objeto. Uma propriedade que oferece o acessador get pode ser usada para acessar o conteúdo de um objeto. Uma propriedade que oferece o acessador set pode ser usada para modificar o conteúdo de um objeto.

Um acessador get é um método que acessa o conteúdo de um objeto mas não o modifica. No caso mais simples, um acessador get simplesmente retorna o valor de um dos campos encapsulados (definidos dentro) no objeto. Em geral, esse acessador realiza algumas operações usando os campos desde que essas operações não modifiquem o valor de nenhum campo.

Um acessador set é um método que modifica um objeto. Um método que modifica um objeto é também conhecido como um modificador (*mutator*). No caso mais simples, um acessador set altera um único campo de um objeto. Em geral, esse acessador pode modificar qualquer número de campos do objeto.

O programa 4.3 define mais duas propriedades da classe Complex – R e Theta. Essas propriedades fornecem acessadores get e set que acessam o número complexo usando coordenadas polares (forma trigonométrica).

```
1.  public class Complex
2.  {
3.      private double real;
4.      private double imag;
5.
6.      public double R
7.      {
8.          get { => Math.Sqrt(real * real + imag * imag); }
9.          set
10.         {
11.             double theta = Theta;
12.             real = value * Math.Cos(theta);
13.             imag = value * Math.Sin(theta);
14.         }
15.     }
```

```
15.     public double Theta
16.     {
17.         get { => Math.Atan2(imag, real); }
18.         set
19.         {
20.             double r = R;
21.             real = r * Math.Cos(value);
22.             imag = r * Math.Sin(value);
23.         }
24.     }
//...
```

Programa 4.3: Propriedades R e Theta da classe Complex

Definindo acessadores adequados, é possível ocultar a implementação da classe do usuário dessa classe. Considere os comandos abaixo:

```
Console.WriteLine(c.real);
Console.WriteLine(c.Real);
```

O primeiro comando depende da implementação da classe Complex. Se modificarmos a implementação original dessa classe (que utilize coordenadas retangulares) para uma que usa coordenadas polares, então o primeiro comando acima também deverá ser alterado. Por outro lado, o segundo comando não necessita de modificações, desde que reimplementemos a propriedade Real quando alternarmos para coordenadas polares.

Controle de acesso a membros de classe

Cada membro de uma classe, seja ele campo ou método, tem um *atributo de controle de acesso* que afeta a maneira pela qual esse membro pode ser acessado. Os membros de uma classe podem ser **private**, **public**, **protected**, **internal**, ou **protected internal**. Por exemplo, os campos **real** e **imag** declarados no programa 4.1 são ambos **private**. Membros **private** podem ser usados apenas pelos métodos da classe dentro da qual estão declarados.

Por outro lado, membros **public** de uma classe podem ser usados por qualquer método em qualquer classe. Todas as operações definidas nos programas 4.1, 4.2 e 4.3 foram declaradas como **public**.

Na realidade, a parte pública de uma classe define a interface para essa classe e a parte privativa encapsula a implementação da classe. Fazendo a implementação de uma classe como privativa, nós garantimos que o código que utilize essa classe dependa apenas da interface e não da implementação da classe. Além disso, podemos modificar a implementação da classe sem afetar o código de qualquer aplicação que use essa classe.

Membros **protected** são similares aos membros **private**. Dessa forma, podem ser usados pelos métodos da classe. Além disso, membros **protected** podem também ser usados pelos métodos de todas as classes derivadas da classe em que o membro é declarado. Mais adiante no texto discutiremos a categoria **protected** com mais detalhes.

Um membro **internal** pode ser acessado por qualquer método no mesmo programa onde o membro é declarado. Por fim, um membro **protected internal** pode ser acessado por métodos na classe em que ele foi declarado e por métodos das classes derivadas da classe onde foi declarado e, também, por qualquer método no mesmo programa em que ele foi declarado.

Classes Aninhadas (nested classes)

Em C# é possível definir uma classe *dentro* de outra. A classe definida dentro de outra é chamada de classe aninhada ou *nested class*, em inglês.

Considere o seguinte fragmento de código C#:

```
public class A
{
    int y;

    public static class B
    {
        int x;

        void F() { }
    }
}
```

Esse fragmento define a classe A, que contém a classe aninhada B.

Uma classe aninhada se comporta como qualquer outra classe “externa”. Ela pode conter campos e métodos, e pode ser instanciada como:

```
A.B obj = new A.B();
```

Este comando cria uma nova instância da classe aninhada B. A partir dessa instância, podemos chamar o método F da maneira usual:

```
obj.F();
```

Note que não é necessário que uma instância da classe A exista quando estamos criando a instância da classe aninhada. Similarmente, instanciar a classe exterior A não cria quaisquer instâncias da classe interior B.

Os métodos de uma classe aninhada podem acessar todos os membros (campos e métodos) da classe aninhada mas somente membros estáticos (campos ou métodos) da classe exterior. Portanto, F pode acessar o campo x, mas não pode acessar o campo y de A.

Herança e Polimorfismo

Derivação e Herança

Classes derivadas são um recurso bastante útil de C# porque elas permitem ao programador definir novas classes pela extensão de classes já existentes. Com elas, o programador por explorar as características comuns que existem entre as classes de um programa. Classes distintas podem, assim, compartilhar valores, operações e interfaces.

Derivação é a definição de uma nova classe pela extensão de uma classe já existente. A nova classe é chamada de *classe derivada* e a classe existente, da qual ela é derivada, é chamada de *classe base*. Em C# só pode haver uma única classe base para cada classe derivada, ou seja, C# só fornece *herança simples*, como Java e Delphi.

Considere a classe Pessoa definida no programa 4.4 e a classe Pai definida no programa 4.5. Como pais também são pessoas, a classe Pai é derivada da classe Pessoa. Derivação em C# é indicada pelo símbolo “:” seguido do nome da classe base na declaração da classe derivada.

```
1.     public class Pessoa
2.     {
3.         public enum Sex { MASCULINO, FEMININO };
4.         protected string nome;
5.         protected Sex sexo;
6.
7.         public Pessoa(string nome, Sex sexo)
8.         {
```

```
8.         this.nome = nome;
9.         this.sexo = sexo;
10.    }

11.   public override string ToString()
12. {
13.     return nome;
14. }
15. }
```

Programa 4.4 – Classe Pessoa

```
1. public class Pai : Pessoa
2. {
3.   protected Pessoa[] filhos;

4.   public Pai(string nome, Sex sexo,
5.   params Pessoa[] filhos)
6.   : base(nome, sexo)
7.   {
8.     this.filhos = filhos;
9.   }

10.  public Pessoa getFilho(int i)
11.  {
12.    return filhos[i];
13.  }

14.  public override string ToString()
15.  {
16.    // ....
17.  }
18. }
```

Programa 4.5 – Classe Pai, derivada da classe Pessoa

Uma classe derivada *herda* todos os membros de sua classe base. Portanto, a classe derivada contém todos os campos contidos na classe base e, também, suporta todas as mesmas operações fornecidas pela classe base. Por exemplo, considere as declarações de variáveis a seguir:

```
Pessoa p = new Pessoa();
Pai     q = new Pai();
```

Já que p é uma Pessoa, ele possui os campos nome e sexo e o método ToString. Além disso, como Pai é derivado de Pessoa, então o objeto q também tem os campos nome e sexo e o método ToString().

Uma classe derivada pode estender a classe base em diversas maneiras : novos campos podem ser definidos, novos métodos podem ser adicionados, e métodos existentes podem ser *sobrepostos (overriden)*. Por exemplo, a classe Pai adiciona o campo filhos e o método getFilho.

Se um método é definido em uma classe derivada que tem exatamente a mesma assinatura (interface : nome e tipos de argumentos) que um método na classe base, o método na classe derivada sobrepõe-se ao mesmo método na classe base. Por exemplo, o método ToString na classe Pai sobrepõe-se ao método ToString na classe Pessoa. Portanto, p.ToString() chama Pessoa.ToString, enquanto q.ToString() chama Pai.ToString. Note que C# exige o uso da palavra reservada override na declaração de um método que se sobrepõe um método herdado.

Uma instância de uma classe derivada pode ser usada em qualquer lugar no programa onde uma instância da classe base possa ser usada. **Por exemplo, isto significa que um Pai pode ser**

passado como um parâmetro real para um método em que o parâmetro formal seja uma Pessoa.

Também é possível atribuir um objeto de classe derivada para uma variável da classe base, como abaixo:

```
Pessoa p = new Pai();
```

No entanto, ao fazer isso, não é possível chamar o método `p.GetFilho(...)`, pois `p` é uma Pessoa e Pessoa não necessariamente é um Pai.

Traduzido de [http://msdn.microsoft.com/en-us/library/aa645765\(v=VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa645765(v=VS.71).aspx)

Observe a linha 5 do programa 4.5. Ela possui a palavra **params** na declaração de um vetor filhos. Um parâmetro declarado com o modificador **params** é um vetor de parâmetros. Se uma lista de parâmetros formais inclui um vetor de parâmetros, este deve ser o último parâmetro na lista e deve ser um tipo de vetor unidimensional. Por exemplo, os tipos `string[]` e `string[][]` podem ser usados como o tipo de um vetor de parâmetros, mas o tipo `string[,]` não pode. Não é possível combinar o modificador `params` com os modificadores **ref** e **out**.

Um vetor de parâmetros permite que os argumentos sejam especificados em uma de duas maneiras na chamada ao método:

- O argumento associado a um vetor de parâmetros pode ser uma expressão simples de um tipo que seja conversível implicitamente ao tipo do vetor de parâmetros. Neste caso, o vetor de parâmetros age precisamente como um parâmetro por valor.
- Alternativamente, a chamada pode especificar zero ou mais argumentos para o vetor de parâmetros, onde cada argumento é uma expressão de um tipo que seja implicitamente conversível ao tipo de elementos do vetor de parâmetros. Neste caso, a chamada cria uma instância do tipo do vetor de parâmetros com um comprimento correspondente ao número de argumentos, inicializa os elementos da instância do vetor com os valores dados nos argumentos e utiliza a recém-criada instância como o argumento real.

```
Pessoa[] vetFilhos = new Pessoa() [10];
...
vetFilhos[0] = new Pessoa("Maria", FEMININO);
vetFilhos[1]=new Pessoa("Abel", MASCULINO);
Pai Joao = new Pai("João", MASCULINO, vetFilhos);
```

Ou

```
Pai Joao = new Pai("João", MASCULINO,
new Pessoa("Maria", FEMININO),
new Pessoa("Abel", MASCULINO),
new Pessoa("Carla", FEMININO)
);
```

Exceto por permitir um número variável de argumentos em uma chamada de método, um vetor de parâmetros é exatamente equivalente a um parâmetro por valor do mesmo tipo.

O exemplo:

```
using System;
class Test
{
    static void F(params int[] args) {
        Console.WriteLine("Array contains {0} elements:", args.Length);
        foreach (int i in args)
```

```
        Console.WriteLine(" {0}", i);
        Console.WriteLine();
    }
    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

produz o resultado

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

A primeira chamada de F simplesmente passa o vetor como um parâmetro por valor. A segunda chamada de F automaticamente cria um vetor int[] com quatro elementos com os valores dados e passa essa instância de vetor como um parâmetro por valor. De forma semelhante, a terceira chamada de F cria um vetor int[] com zero elementos e passa essa instância como um parâmetro por valor. A segunda e terceira chamadas são exatamente equivalentes ao código abaixo:

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

Quando realiza uma resolução de sobrecarga, um método com um vetor de parâmetros pode ser aplicável tanto em sua forma normal quanto em sua forma expandida. A forma expandida de um método é disponível somente se a forma normal do método não é aplicável e somente se um método com a mesma assinatura que a forma expandida ainda não foi declarado no mesmo tipo.

O exemplo

```
using System;
class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }
    static void F() {
        Console.WriteLine("F()");
    }
    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }
    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}
```

produz o resultado

```
F();
F(object[]);
```

```
F(object,object);
F(object[]);
F(object[]);
```

No exemplo, duas das possíveis formas expandidas do método com um vetor de parâmetros já estão incluídas na classe como métodos regulares. Essas formas expandidas são, portanto, não consideradas quando da realização da resolução de sobrecarga e, assim, as chamadas do primeiro e terceiro métodos selecionam os métodos regulares. Quando uma classe declara um método com um vetor de parâmetros, não é incomum também incluir algumas das formas expandidas como métodos regulares. Fazendo isso é possível evitar a alocação de uma instância do vetor que ocorre quando uma forma expandida de um método com um vetor de parâmetros é chamado.

Quando o tipo de um vetor de parâmetros é `object[]`, uma ambiguidade potencial surge entre a forma normal do método e a forma expandida uma um único parâmetro de tipo `object`. A razão para a ambiguidade é que um `object[]` é, por si mesmo, implicitamente conversível ao tipo `object`. No entanto, a ambiguidade não apresenta nenhum problema, visto que ela pode ser resolvida pela inserção de um cast se necessário.

O exemplo

```
using System;
class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.WriteLine(o.GetType().FullName);
            Console.Write(" ");
        }
        Console.WriteLine();
    }
    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

produz o resultado

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

Na primeira e última chamadas de `F`, a forma normal de `F` é aplicável, porque uma conversão implícita existe do tipo de argumento para o tipo do parâmetro (ambos são do tipo `object[]`). Assim, a resolução da sobrecarga seleciona a forma normal de `F`, e o argumento é passado como um parâmetro regular por valor. Na segunda e terceira chamadas, a forma normal de `F` não é aplicável porque nenhuma conversão implícita existe do tipo do argumento para o tipo do parâmetro (o tipo `object` não pode ser implicitamente convertido para o tipo `object[]`). No entanto, a forma expandida de `F` é aplicável, de forma que esta é selecionada pela resolução de sobrecarga. Como resultado, um `object[]` de um elemento é criado pela chamada e o único elemento do vetor é iniciado com o valor dado do argumento (o qual, por si só, é uma referência a um `object[]`).

Derivação e Controle de Acesso

Membros de uma classe podem ser declarados como **private**, **public** ou **protected**. Como explicado (Controle de acesso a membros de classe), membros **private** são acessíveis apenas pelos métodos da classe em que são declarados. Especificamente, isto significa que os métodos de uma classe derivada não podem acessar os membros privativos da classe base mesmo que a classe derivada tenha herdado esses membros! Por outro lado, se tornarmos públicos os membros da classe base, todas as classes poderão acessar esses membros diretamente, não apenas as classes derivadas. Essa abordagem aumenta o grau de acoplamento da classe, permitindo acessos indevidos pelas aplicações que usam essas classes.

É aqui, portanto, que se utilize o controle de acesso **protected**. Membros declarados como **protected** podem ser usados pelos métodos da classe em que são declarados, bem como pelos métodos de todas as classes derivadas da classe onde o membro foi declarado.

Polimorfismo

Polimorfismo literalmente significa “tendo muitas formas”. O polimorfismo ocorre quando um conjunto de classes distintas compartilham uma interface comum. Devido ao fato de as classes derivadas serem distintas, suas implementações podem diferir. No entanto, já que as classes derivadas compartilham uma interface comum, instâncias dessas classes são usadas exatamente da mesma maneira.

Interfaces

Considere um programa para criar desenhos simples. Suponha que o programa fornece um conjunto de objetos gráficos primitivos (básicos), tais como círculos, retângulos e quadrados. O usuário do programa seleciona os objetos desejados e então invoca comandos para desenhá-los, apagá-los ou movê-los. Idealmente, todos os objetos gráficos suportam as mesmas operações. Mesmo assim, a maneira como cada operação é implementada varia de um tipo de objeto gráfico para outro.

Programamos esse conceito da seguinte maneira: primeiramente, definimos uma *interface* C# que representa as operações comuns fornecidas por todos os objetos gráficos. Uma interface C# declara um conjunto de métodos. Um objeto que suportar essa interface deve fornecer o código dos métodos declarados (previstos) na interface.

O programa 4.6 define a interface PrimitivasGraficas composta por três métodos, Desenhar, Apagar e MoverPara.

```
interface PrimitivasGraficas
{
    void Desenhar();
    void Apagar();
    void MoverPara(Ponto delta);
}
```

Programa 4.6 – Interface PrimitivasGraficas

O método Desenhar é chamado para desenhar um objeto gráfico. O método Apagar é chamado para apagar um objeto gráfico. O método MoverPara é usado para mover um objeto para uma posição específica da área de desenho. O argumento do método MoverPara é uma estrutura (struct) do tipo Ponto. O programa 4.7. define a struct Ponto que representa uma posição na area de desenho.

```
public struct Ponto
{
    int x;
    int y;
```

```
public Ponto(int x, int y)
{
    this.x = x;
    this.y = y;
}
//...
```

Programa 4.7 – Estrutura Ponto

Métodos e Classes Abstratos

Considere a classe abstrata ObjetoGrafico definida no programa 4.8. Essa classe *implementa* a interface PrimitivasGraficas. Isso é indicado pelo símbolo “:” seguido pelo nome da interface na declaração da classe abstrata.

```
public abstract class ObjetoGrafico : PrimitivasGraficas
{
    protected Ponto centro;

    public ObjetoGrafico(Ponto p)
    {
        this.centro = p;
    }

    public abstract void Desenhar();

    public virtual void Apagar()
    {
        SetPenColor(COR_DE_FUNDO);
        Desenhar();
        SetPenColor(COR_DE_FRENTES);
    }

    public virtual void MoverPara(Ponto p)
    {
        Apagar();
        centro = p;
        Desenhar();
    }
}
```

Não implementa todos os seus métodos

Este método será implementado por uma classe derivada

Em C#, para indicar que um método poderá ser escrito com a mesma assinatura em uma classe derivada, utilizamos a palavra reservada **virtual** na classe base e a palavra reservada **override** na classe derivada, quando esta for implementar a nova forma do método.

Programa 4.8 – Classe ObjetoGrafico

A classe ObjetoGrafico tem um único campo, centro, que é um Ponto representando a posição no desenho do ponto central do objeto gráfico. O construtor para a classe ObjetoGrafico tem como argumento um Ponto e inicia o campo centro com o valor do argumento.

O programa 4.8 mostra uma possível implementação do método Apagar. Neste caso assumimos que a imagem é desenhada com uma “caneta” imaginária. Assumindo que sabemos como desenhar um objeto gráfico, podemos apagar o objeto mudando a cor da caneta de modo que ela fique igual à cor da área de desenho (COR_DE_FUNDO) e então, redesenhando o objeto.

Uma vez que nós podemos tanto apagar quanto desenhar um objeto, então movê-lo é fácil. Apenas apague o objeto, mude seu ponto central e redesenho-o. Esta é a forma como o método MoverPara foi implementado.

Vimos que a classe ObjetoGrafico fornece implementações para os métodos **Apagar** e **MoverPara**. No entanto, essa classe não fornece a implementação do método Desenhar. Ao invés

disso, o método é declarado como **abstract**. Fizemos isso porque, até que saibamos em que tipo de objeto a classe está sendo usada, não saberemos como desenhar a figura corretamente!

Considere a classe Circulo definida no Programa 4.9. Essa classe **estende** a classe ObjetoGrafico. Portanto, ela herda o campo **centro** e os métodos **Apagar** e **MoverPara**. A classe Circulo apresenta um campo adicional, **raio**, e ela sobrepuja o método **Desenhar**. O corpo do método Desenhar não é mostrado no programa 4.9. No entanto, podemos assumir que ele desenhárá um círculo com o centro e o raio dados.

```
namespace Grafico
{
    class Circulo : ObjetoGrafico
    {
        protected int raio;
        public Circulo(Ponto p, int r) : base(p)
        {
            raio = r;
        }

        public override void Desenhar()
        {
            throw new NotImplementedException(); // colocado pelo Visual Studio
        }
    }
}
```

Indica que se está implementando um método virtual ou abstract da classe base

Programa 4.9 – Classe Circulo, herança de ObjetoGrafico

Usando a classe Circulo definida no programa 4.9, podemos escrever código como o que se segue:

```
Circulo c = new Circulo(new Ponto(0, 0), 5);
c.Desenhar();
c.MoverPara(new Ponto(10, 10));
c.Apagar();
```

Esta sequência de código declara um objeto círculo com seu centro inicialmente posicionado na coordenada (0,0) e raio 5. O círculo é, então, desenhado, movido para a posição (10,10), e, posteriormente, apagado.

O programa 4.10 define a classe Retangulo e o programa 4.11 define a classe Quadrado. A classe Retangulo é também uma extensão (herança) da classe ObjetoGrafico. Portanto, ela herda o campo centro e os métodos Apagar e MoverPara. A classe Retangulo possui dois campos adicionais, altura e largura, e sobrepuja o método Desenhar. O corpo desse método, no entanto, não é mostrado no programa 4.10, mas assumimos que ele desenhe um retângulo dados os valores das dimensões e do ponto central da figura.

```
namespace Grafico
{
    class Retangulo : ObjetoGrafico
    {
        protected int altura;
        protected int largura;
        public Retangulo(Ponto p, int a, int l) : base(p)
        {
            altura = a;
            largura = l;
        }

        public override void Desenhar()
```

```
{  
    throw new NotImplementedException(); // colocado pelo Visual Studio  
}  
}  
}
```

Programa 4.10 – Classe Retangulo, herança de ObjetoGrafico

Já a classe Quadrado estende a classe Retangulo. Nenhum novo campo ou método são declarados, pois aqueles herdados de ObjetoGrafico ou de Retangulo são suficientes. O construtor simplesmente se responsabiliza por garantir que a altura e a largura do quadrado sejam iguais!

```
namespace Grafico  
{  
    class Quadrado : Retangulo  
    {  
        public Quadrado(Ponto p, int l) : base(p, l, l)  
        {  
        }  
    }  
}
```

Programa 4.11 – Classe Quadrado, herança de Retangulo

Resolução de Métodos

Considere a seqüência de instruções a seguir:

```
ObjetoGrafico g1 = new Circulo(new Ponto (0,0), 5);  
ObjetoGrafico g2 = new Quadrado(new Ponto (0,0), 5);  
g1.Desenhar ();  
g2.Desenhar ();
```

O comando g1. Desenhar() chama Circulo.Desenhar enquanto g2.Desenhar() chama Retângulo.Desenhar()..

Isso funciona como se cada objeto de uma classe soubesse os métodos reais a serem invocados quando um método é chamado naquele objeto. Por exemplo, um círculo sabe chamar Circulo.Desenhar, ObjetoGrafico.Apagar e ObjetoGrafico.MoverPara, enquanto um quadrado sabe chamar Retângulo.Desenhar, ObjetoGrafico.Apagar e ObjetoGrafico.MoverPara.

Desta maneira, C# assegura que o método correto é chamado verdadeiramente, independentemente de como o objeto é acessado.

Considere a sequência a seguir:

```
Quadrado q = new Quadrado (new Ponto(0,0), 5);  
Retangulo r = q;  
ObjetoGrafico g = r;
```

Aqui, q, r e g se referem todos ao mesmo objeto, pois apenas um foi instanciado. Isso acontece mesmo que essas variáveis sejam de tipos diferentes. No entanto, devido ao fato do objeto referenciado ser um Quadrado, q.Desenhar(), r.Desenhar() e g.Desenhar() todos invocam Retangulo.Desenhar()..

Classes Abstratas e Classes Concretas

Em C# uma **classe abstrata** é uma classe que não fornece a implementação de todos os seus métodos. Uma classe deve ser declarada com a palavra **abstract** se quaisquer de seus métodos são abstratos. Por exemplo, a classe ObjetoGrafico definida no programa 4.8 é declarada **abstract** porque seu método Desenhar é abstrato.

Uma classe abstrata será usada como classe base para outras classes que dela serão derivadas. A classe derivada deverá fornecer implementações para os métodos que não foram implementados na classe base. Uma classe derivada que implemente todos os métodos abstratos da classe base é chamada de **classe concreta**.

Em C# não é possível instanciar uma classe abstrata. Por exemplo, a declaração a seguir é ilegal:

```
ObjetoGrafico g = new ObjetoGrafico (new Ponto(0,0)); // Errado.
```

Se pudéssemos declarar g desta maneira, poderíamos também invocar o método não existente g.Desenhar().

Abstração de Algoritmos

Classes abstratas podem ser usadas de muitas formas interessantes. Um dos paradigmas mais úteis é a utilização de uma classe abstrata para abstração de algoritmos. Os métodos Apagar e MoverPara definidos no programa 4.8 são exemplos dessa técnica.

Os métodos Apagar e MoverPara são implementados na classe ObjetoGrafico. Os algoritmos implementados foram projetados para funcionar em qualquer classe concreta derivada de ObjetoGrafico, seja ela Circulo, Retangulo ou Quadrado. Nós escrevemos algoritmos que funcionam sem que se tenha de levar em conta a classe real do objeto. Portanto, tais algoritmos são chamados algoritmos abstratos.

Algoritmos abstratos geralmente invocam métodos abstratos. Por exemplo, tanto MoverPara quanto Apagar acabam chamando Desenhar para fazer a maioria de seu trabalho real. Neste caso, as classes derivadas deverão herdar os algoritmos abstratos MoverPara e Apagar e sobrepor o método abstrato Desenhar. Dessa maneira, as classes derivadas configuram o comportamento do algoritmo abstrato pela sobreposição dos métodos apropriados. O mecanismo de resolução de métodos do C# assegura que o método “correto” sempre é chamado.

Herança Múltipla

Em C# uma classe pode ser derivada de apenas uma única classe base. Portanto, a declaração abaixo não é permitida:

```
class A {}  
class B {}  
class C : A, B // Errado!  
{  
}
```

No entanto, é possível para uma classe estender uma classe base e implementar uma ou mais interfaces (abaixo supomos que D e E são interfaces, não classes):

```
class A {}  
interface D {}  
interface E {}  
class C : A, D, E  
{  
}
```

A classe derivada C herda os membros da classe base A e implementa todos os métodos definidos nas interfaces D e E.

É possível usar derivação na definição de interfaces. E, em C#, é possível para uma interface estender mais que uma interface base:

```
interface E {}  
interface F {}  
interface D : E, F  
{  
}
```

Neste caso, a interface derivada D conterá todos os métodos herdados de E e F, bem como quaisquer novos métodos declarados no corpo de D.

Informação de Tipos em tempo de execução e Casts

Considere as declarações abaixo, que fazem uso das classes Retangulo e Quadrado definidas nos programas 4.10 e 4.11:

```
Retangulo r = new Retangulo(new Ponto(0,0), 5, 10);  
Quadrado q = new Quadrado(new Ponto(0,0), 15);
```

Obviamente, a atribuição

```
r = q;
```

é válida porque Quadrado é derivada de Retangulo. Ou seja, já que um Quadrado é um Retângulo, podemos atribuir q a r.

Por outro lado, a atribuição

```
q = r; // Errado!
```

não é válida porque uma instância de Retangulo não é, necessariamente, um Quadrado.

Considere agora as seguintes declarações:

```
Retangulo r = new Quadrado(new Ponto(0,0), 20);  
Quadrado q;
```

A atribuição q = r ainda é inválida porque r é um Retangulo, e um Retangulo não é necessariamente um quadrado, mesmo que, neste caso, ele realmente o seja!

Para realizar essa atribuição, é necessário converter o tipo de r, de um Retangulo para um Quadrado. Em C# isto é feito usando um operador de cast:

```
q = (Quadrado)r;
```

A linguagem comum de tempo de execução (CLR) do C# verifica durante a execução que r realmente se refere a um Quadrado e se ele não o faz, a operação dispara uma exceção do tipo ClassCastException. (exceções serão discutidas mais à frente).

Para determinar o tipo do objeto ao qual r se refere, devemos usar a *informação de tipo em tempo de execução* (run-time Type Information ou RTTI). Em C# o operador **is** pode ser usado para verificar se um objeto específico é uma instância de alguma classe. Dessa forma, podemos determinar a classe de um objeto da maneira abaixo:

```
if (r is Quadrado)
    q = (Quadrado)r;
```

Este código não dispara uma exceção porque o operador de cast somente é usado quando r realmente é um Quadrado.

Alternativamente, podemos usar o operador **as** para realizar a conversão, como vemos abaixo:

```
q = r as Quadrado;
```

O operador **as** retorna **null** (e não dispara uma exceção) se o objeto ao qual r se refere não é um Quadrado.

Classes Parciais

O conceito de classes parciais presente no .Net Framework serve para dividir o código de nossas estruturas em dois ou mais arquivos fonte. As estruturas passíveis a essa divisão são classes, structs e interfaces.

As classes parciais nos possibilitam uma melhor organização do nosso código, visto que podemos distribuir o código de uma classe grande em vários arquivos (por exemplo, podemos escrever os atributos, propriedades e construtores da classe em um fonte e os métodos e funções em outro fonte).

Abaixo veremos um exemplo de uma classe chamada Coordenada que foi dividida em dois arquivos fonte: Coordenada.cs e CoordenadaMetodos.cs:

Coordenada.cs:	CoordenadaMetodos.cs:
<pre>using System; using System.Collections.Generic; using System.Linq; using System.Text; namespace NamespacesDemo { public partial class Coordenada { private int _x; private int _y; public int X { get { return _x; } set { _x = value; } } public int Y { get { return _y; } set { _y = value; } } } }</pre>	<pre>using System; using System.Collections.Generic; using System.Linq; using System.Text; namespace NamespacesDemo { public partial class Coordenada { public void PrintCoordenadas() { Console.WriteLine(this._x.ToString() + " " + this._y); } } }</pre>

Como podemos verificar nos códigos acima, separamos os atributos e propriedades da classe Coordenada no arquivo fonte Coordenada.cs e o método PrintCoordenadas no arquivo fonte CoordenadaMetodos.cs.

Para indicar que uma classe está dividida em mais de um arquivo fonte, utilizamos a palavra reservada **partial**.

Ao compilar o código, o .Net acaba juntando o conteúdo dos dois arquivos fonte em um só, para chegar ao código completo da classe Coordenada.

Se repararmos nos arquivos criados ao adicionarmos um formem um projeto Windows Forms, percebemos que o Visual Studio na verdade separa o código em dois arquivos: um fonte contém o código de design do form, que vai sendo criado pelo Visual Studio ao customizarmos suas propriedades pelo editor visual e outro fonte contém o restante do código do form (implementação de eventos, métodos, etc.).

Leia mais em: <http://www.linhadecodigo.com.br/artigo/2320/namespaces-classes-parciais-e-metodos-virtuais-em-csharp.aspx#ixzz2MVd8DvVD>

Exceções

Por vezes, situações inesperadas acontecem durante a execução de um programa. Programadores cuidadosos escrevem código que detecta erros e lida com eles de forma apropriada. No entanto, um algoritmo simples pode se tornar ininteligível quando verificações de erros são adicionadas a ele porque o código de verificação de erros pode obscurecer a operação normal do algoritmo.

Exceções fornecem uma forma limpa de detectar e tratar situações inesperadas. Quando um programa detecta um erro, ele dispara uma exceção. Quando uma exceção é disparada, o controle é transferido para um *tratador de exceção* apropriado. Ao definir um método que captura a exceção, o programador pode escrever código para tratar o erro.

Em C#, uma exceção é um objeto. Todas as exceções em C# são derivadas da classe base chamada System.Exception. Por exemplo, considere a classe A definida no Programa 4.12. Como a classe A estende a classe System.Exception, A é uma exceção que pode ser *disparada*:

```
public class Excecoes
{
    public class A : System.Exception
    {....}

    static void F() {
        throw new A();
    }

    static void G() {
        try
        {
            F();
        }
        catch (A exception)
        {
            MessageBox.Show("Exceção");
        }
    }
}
```

Programa 4.12 – Usando exceções em C#

Um método dispara uma exceção pelo uso do comando **throw**: o comando **throw** é similar a um comando **return**. Um comando **return** representa o término normal de um método e o objeto

retornado combina com o valor de retorno do método. Já um comando **throw** representa o término anormal de um método e o objeto disparado representa o tipo do erro encontrado. O método F no programa 4.12 dispara uma exceção do tipo A.

Tratadores de exceções são definidos pelo uso de um bloco **try**: o corpo de um bloco **try** é executado até que uma exceção seja disparada ou até que o bloco termine normalmente.. Um ou mais tratadores de exceções seguem um bloco **try**. Cada tratador de exceção consiste de uma cláusula **catch** que especifica as exceções a serem capturadas, e um bloco de código, que é executado quando a exceção ocorre. Quando o corpo do bloco **try** dispara uma exceção para a qual um tratador é definido, o controle é transferido para o corpo do tratador daquela exceção.

Neste exemplo, a exceção disparada pelo método F é capturada pelo método G. Em geral, quando uma exceção é disparada, a cadeia de métodos chamados é pesquisada em ordem reversa (do chamador para o chamado) para encontrar o comando catch mais próximo que combine com o tipo da exceção disparada. Quando um programa dispara uma exceção que não é capturada, o programa é finalizado.

Programação Genérica

Um dos problemas com POO é uma característica chamada “code bloat” (significa algo como “inchamento de código” ou “entumescimento de código”, mas procure no Google ou em www.devx.com!). Um tipo de code bloat ocorre quando você tem que sobrepor um método, ou um conjunto de métodos, para levar em conta todos os possíveis tipos de dados dos parâmetros dos métodos.

Uma solução para o code bloat é a capacidade de um valor assumir múltiplos tipos de dados, enquanto somente se providencia uma definição desse valor. Esta técnica é chamada de Programação Genérica.

Um programa genérico fornece um “armazenador” que é preenchido por um tipo específico de dados em tempo de compilação. Esse “armazém” é representado por um par de sinais < e >, com um identificador colocado entre os sinais. Abaixo temos um exemplo:

Um primeiro exemplo canônico para programação genérica é a função Swap, que troca o conteúdo de dois valores quaisquer, de quaisquer tipos.

Aqui temos a definição da função genérica Swap em C#:

```
static void Swap<T>(ref T val1, ref T val2)
{
    T aux;
    aux = val1;
    val1 = val2;
    val2 = aux;
}
```

O “armazém” para o tipo de dados é colocado imediatamente após o nome da função. O identificador colocado entre < e > sera usado sempre que um tipo genérico de dados é necessário. A cada um dos parâmetros é associado um tipo genérico de dados, como no caso da variável aux usada para realizar a troca. Abaixo segue um programa que usa esse código:

```
using System;

class chapter1
{

    static void Main()
{
    int num1 = 100;
```

```
int num2 = 200;
Console.WriteLine("num1: " + num1);
Console.WriteLine("num2: " + num2);

Swap<int>(ref num1, ref num2);

Console.WriteLine("num1: " + num1);
Console.WriteLine("num2: " + num2);

string str1 = "Sam";
string str2 = "Tom";
Console.WriteLine("String 1: " + str1);
Console.WriteLine("String 2: " + str2);

Swap<string>(ref str1, ref str2);

Console.WriteLine("String 1: " + str1);
Console.WriteLine("String 2: " + str2);
}

static void Swap<T>(ref T val1, ref T val2)
{
    T aux;
    aux = val1;
    val1 = val2;
    val2 = aux;
}
```

```
static void Swap<T>(ref T val1, ref T val2)
{
    T aux;
```

A saída deste programa é:

```
C:\WINDOWS\system32\cmd.exe
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>ch1
num1: 100
num2: 200
num1: 200
num2: 100
String 1: Sam
String 2: Tom
String 1: Tom
String 2: Sam
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>_
```

Generics não são limitados a definições de funções; você pode também criar classes genéricas. Uma definição de classe genérica conterá um armazenador genérico de tipo depois do nome da classe. Em qualquer momento que o nome da classe seja referenciado na definição, o armazenador de tipo deve ser fornecido. A definição de classe a seguir demonstra como criar uma classe genérica:

```
public class NoLista<T>
{
    T dados;
    NoLista<T> proximo;

    public NoLista(T dados, NoLista<T> prox)
    {
        this.dados = dados;
        proximo = prox; // nome de parâmetro difere de nome de campo, não usa this
```

}

Esta classe pode ser usada como abaixo:

```
NoLista<string> node1 = new NoLista <string>("Mike", null);
NoLista<string> node2 = new NoLista <string>("Raymond", node1);
```

Usaremos a classe `NoLista` em várias das estruturas de dados que examinaremos nesta disciplina.

Enquanto o uso de programação genérica pode ser muito útil, C# fornece uma biblioteca de estruturas de dados genéricas já prontas para uso. Essas estruturas de dados são encontradas no namespace `System.Collections.Generic`.

5. Gráficos em C#

Para incrementar nossos projetos práticos, aprenderemos as ferramentas de C# para desenhos de objetos bidimensionais e para controlar cores e fontes. O texto a seguir foi extraído do capítulo 16 do livro C# - Como Programar, de Deitel & Deitel.

C# suporta gráficos que permitem aos programadores aprimorar visualmente as suas aplicações Windows. A Biblioteca de Classes do Framework (FCL) contém muitos recursos de desenho sofisticados como parte do namespace **System.Drawing** e de outros namespaces que compõem o recurso GDI+ do .Net Framework.

GDI+, que é uma extensão da Interface de Dispositivos Gráficos (Graphical Device Interface) é uma interface de programação de aplicações (API) que fornece classes para criar gráficos vetoriais de 2 dimensões, que é uma maneira de descrever gráficos que permite que eles sejam facilmente manipulados com técnicas de alto desempenho. GDI+ também permite a manipulação de imagens e de fontes.

Esse recurso expande o GDI padrão do Windows simplificando o modelo de programação e introduzindo diversos novos recursos, tais como caminhos gráficos, suporte a formatos estendidos de arquivos de imagens e alpha blending (mistura de cores e objetos gráficos). Usando a API do GDI+, programadores podem criar imagens sem se preocupar com os detalhes específicos de plataforma de seu hardware gráfico.

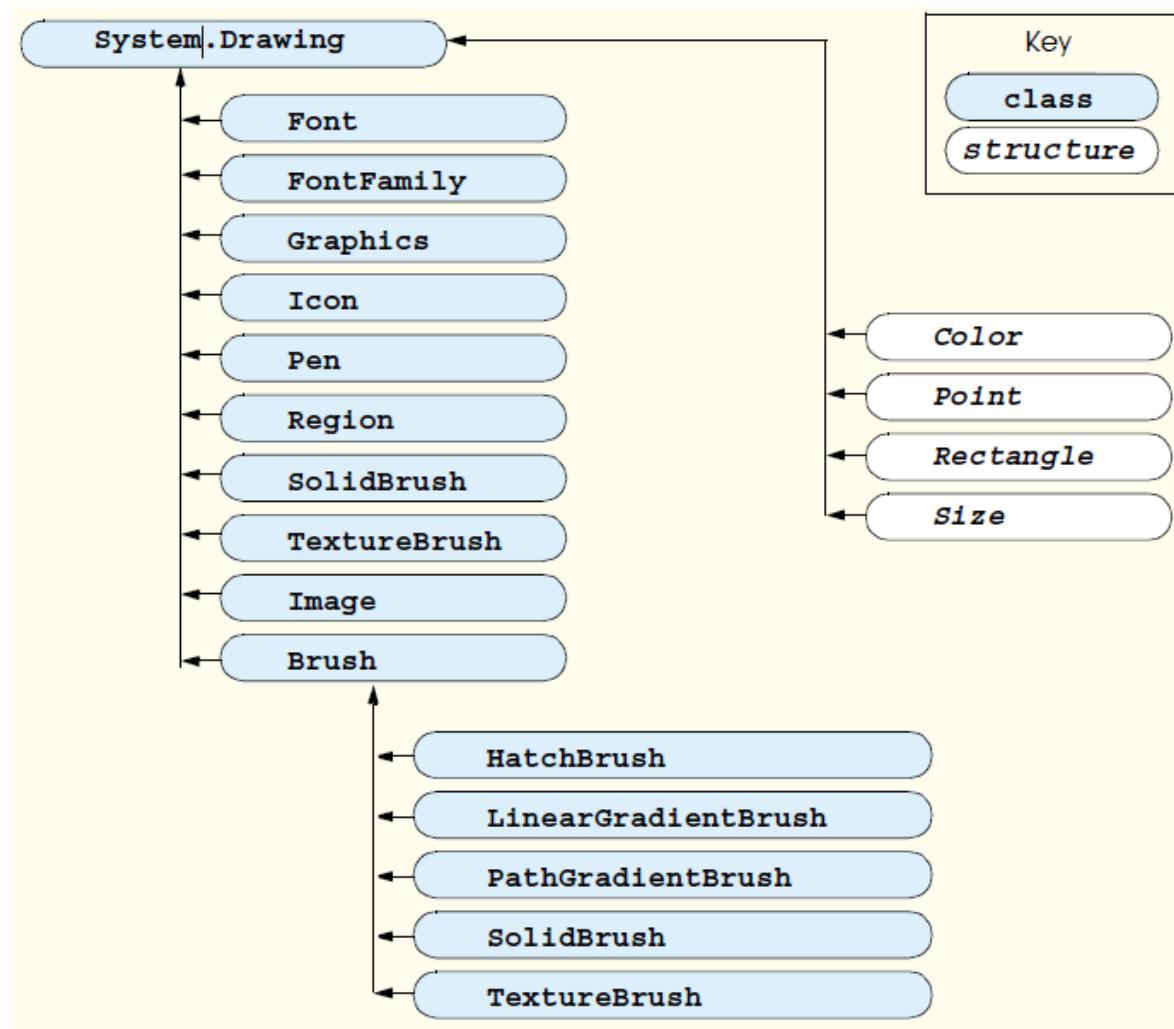


Figura 1 - parte da hierarquia de classes **System.Drawing**

A figura 1 exibe uma parte da hierarquia de classes do namespace System.Drawing, que inclui várias das classes e estruturas gráficas básicas que estudaremos. Os componentes mais comumente usados da GDI+ residem nos namespaces *System.Drawing* e *System.Drawing.Drawing2D*. A classe *Graphics* contém métodos usados para desenhar strings, linhas, retângulos e outras formas geométricas.

Essas formas são desenhadas em um controle de um formulário.

Os métodos de desenho da classe *Graphics* geralmente exigem um objeto *Pen* ou *Brush* para formar um objeto especificado. *Pen* desenha bordas de formas geométricas, enquanto *Brush* desenha objetos sólidos.

A structure *Color* contém inúmeras propriedades estáticas, que configuram as cores dos diversos componentes/gráficos, além de métodos que permitem criar novas cores.

A classe *Font* contém propriedades que definem fontes para textos. A classe *FontFamily* contém métodos para obter informações sobre fontes de texto.

Para começar a desenhar em C#, primeiro devemos entender o sistema de coordenadas do GDI+, através da figura 2. Esse sistema de coordenadas é um esquema que permite identificar cada ponto da tela. Por padrão, o canto superior esquerdo de um componente GUI (Graphical User Interface), tais como um Panel ou um Form, tem as coordenadas (0, 0). Um par coordenado possui tanto a componente de coordenada x (componente horizontal da coordenada) quanto um componente de coordenada y (componente vertical da coordenada). A coordenada x de um ponto qualquer é a distância horizontal (para a direita) do canto superior esquerdo e esse ponto. A coordenada y é a distância vertical (de cima para baixo) a partir do canto superior esquerdo até o ponto em questão. O eixo-x define cada coordenada horizontal, e o eixo-y define cada coordenada vertical.

Programadores posicionam texto e formas geométricas na tela especificando suas coordenadas (x,y). Unidades de coordenadas são medidas em pixels (Picture elements), que são as menores unidades de resolução em um monitor de vídeo.

O namespace System.Drawing fornece as structures *Rectangle* e *Point*. A structure *Rectangle* define formas e dimensões retangulares. A structure *Point* representa as coordenadas (x, y) de um ponto num plano bi-dimensional.

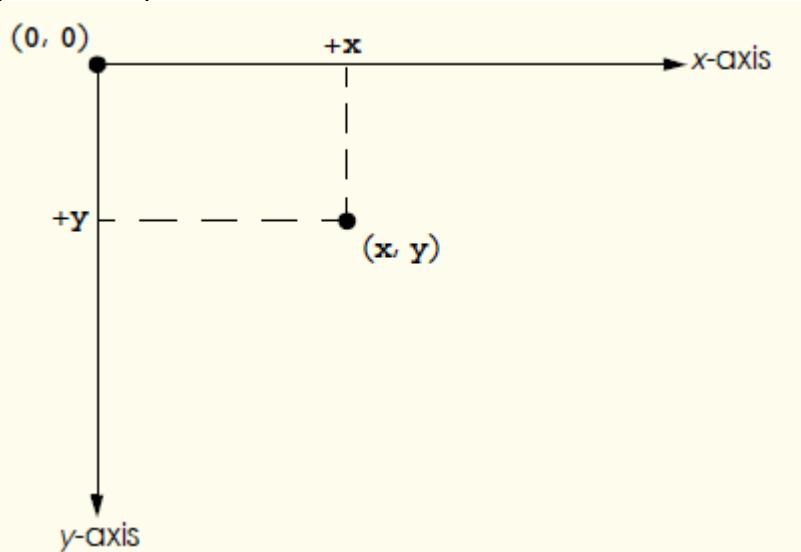


Figura 2 – Sistema de coordenadas GDI+. Unidades são medidas em pixels

Contextos Gráficos e Objetos Gráficos

Um contexto gráfico do C# representa uma superfície de desenho que permite o desenho sobre a tela. Um objeto gráfico gerencia um contexto gráfico através do controle de como a informação é desenhada.

Objetos da classe `Graphics` contém métodos para desenho, manipulação de fontes, manipulação de cores e outras ações relacionadas a gráficos. Cada aplicação Windows que deriva da classe `System.Windows.Forms.Form` herda um tratador do evento virtual `OnPaint` onde a maioria das operações gráficas são realizadas. Os argumentos para o método `OnPaint` incluem um objeto `PaintEventArgs` a partir do qual podemos obter um objeto da classe `Graphics` para o controle que está sendo desenhado. Devemos obter o objeto da classe `Graphics` para cada chamada do método, porque as propriedades dos contextos gráficos que os objetos gráficos representam podem mudar em cada chamada.

O evento `OnPaint` dispara o evento `Paint` do controle. Quando exibirem informações gráficas na área de desenho de um formulário Windows (Form), um programador pode sobrepor (override) o método `OnPaint` para recuperar um objeto da classe `Graphics` proveniente do argumento `PaintEventArgs` ou criar um novo objeto `Graphics` associado com a superfície de desenho apropriada.

Para substituir o método `OnPaint` herdado, usamos a seguinte definição de métodos:

```
protected override void OnPaint( PaintEventArgs e )
```

Em seguida, extraímos o objeto `Graphics` proveniente do argumento `PaintEventArgs` com o comando :

```
Graphics graphicsObject = e.Graphics;
```

A variável `graphicsObject`, da classe `Graphics`, agora estará disponível para o desenho de formas geométricas ou strings.

Chamar o método `OnPaint` dispara o evento `Paint`. Ao invés de sobrepor o método `OnPaint`, podemos adicionar um tratador de evento para o evento `Paint`. Visual Studio.Net gera o tratador de evento `Paint` da seguinte maneira:

```
protected void MyEventHandler_Paint( object sender, PaintEventArgs e )
```

Programadores raramente chamam o método `OnPaint` diretamente, porque o desenho de gráficos é um processo disparado por eventos. Um evento, tais como o ocultamento, cobertura ou descobertura de uma janela, ou a mudança de tamanho de uma janela, chama o método `OnPaint` para o formulário onde isso ocorreu. Da maneira similar, quando qualquer controle (tais como um `TextBox` ou `Label`) é exibido, o programa chama o método `Paint` desse controle.

Se um programador precisa obrigar explicitamente a execução do método `OnPaint`, não deverá chamar esse método diretamente. Ao contrário, deverá chamar o método `Invalidate` (herdado da classe `Control`).

Este método restaura a área de desenho do controle e implicitamente redesenha todos os seus componentes gráficos. C# contém diversos métodos `Invalidate` sobrecarregados que permitem ao programador atualizar partes da área de desenho.

Controles como `Labels` e `Buttons` não têm seus próprios contextos gráficos, mas um pode ser criado para eles. Para desenhar sobre um controle, primeiro crie um objeto gráfico invocando o método `CreateGraphics`.

```
Graphics graphicsObject = controlName.CreateGraphics();
```

Onde `graphicsObject` representa uma instância da classe `Graphics` e `controlName` é qualquer controle.

Com isso, pode-se usar os métodos fornecidos pela classe `Graphics` para desenhar sobre o controle, sem levar em conta a configuração que foi definida para esse controle na janela Properties em tempo de design do formulário.

Desenhando Linhas, Retângulos e Ovais

Cada um dos métodos que veremos abaixo possuem varias versões sobrecarregadas. Quando usamos métodos para desenhar bordas de formas geométricas, usamos versões que recebem um objeto `Pen` e quatro `ints`; quando usamos métodos para desenhar formas geométricas sólidas, usamos versões que recebem um objeto `Brush` e quatro `ints`. Nas duas versões, os dois primeiros argumentos `int` representam as coordenadas do canto superior esquerdo da forma geométrica e da área que o envolve, e os dois últimos argumentos `int` representam a largura e a altura, respectivamente, da forma geométrica.

A lista abaixo resume os métodos de `Graphics` e seus parâmetros.

`DrawLine(Pen p, int x1, int y1, int x2, int y2)`

Traça uma linha de (x_1, y_1) para (x_2, y_2) . **Pen** determina a cor, estilo e espessura da linha.

`DrawRectangle(Pen p, int x, int y, int largura, int altura)`

Traça um retângulo com a largura e altura especificados. O canto superior esquerdo do retângulo estará no ponto (x, y) . **Pen** determina a cor, estilo e borda do retângulo.

`FillRectangle(Brush b, int x, int y, int largura, int altura)`

Desenha um retângulo sólido com a largura e altura especificados. O canto superior esquerdo do retângulo estará no ponto (x, y) . **Brush** determina o padrão de preenchimento interno do retângulo.

`DrawEllipse(Pen p, int x, int y, int largura, int altura)`

Desenha uma elipse inscrita em um retângulo. A largura e altura do retângulo são as especificadas, e o canto superior esquerdo é o ponto (x, y) . **Pen** determina a cor, estilo e borda da elipse.

`FillEllipse(Brush b, int x, int y, int width, int height)`

Desenha uma elipse preenchida inscrita em um retângulo. A largura e altura do retângulo são as especificadas, e o canto superior esquerdo é o ponto (x, y) . **Brush** determina o padrão de preenchimento interno da elipse.

```
1 // Fig. 7.4: LinesRectanglesOvals.cs
2 // Demonstração de linhas, retângulos e ovais.
3
4 using System;
5 using System.Drawing;
6 using System.Collections;
7 using System.ComponentModel;
8 using System.Windows.Forms;
9 using System.Data;
10
11 // desenha formas geometricas sobre o formulário
12 public class LinesRectanglesOvals : System.Windows.Forms.Form
13 {
14     private System.ComponentModel.Container components = null;
15
16     [STAThread]
17     static void Main()
18     {
19         Application.Run( new LinesRectanglesOvals() );
20     }
}
```

```
21
22 // Visual Studio .NET generated code
23
24 protected override void OnPaint(
25             PaintEventArgs paintEvent )
26 {
27 // get graphics object
28 Graphics g = paintEvent.Graphics;
29 SolidBrush brush = new SolidBrush( Color.Blue );
30 Pen pen = new Pen( Color.AliceBlue );
31
32 // create filled rectangle
33 g.FillRectangle( brush, 90, 30, 150, 90 );
34
35 // draw lines to connect rectangles
36 g.DrawLine( pen, 90, 30, 110, 40 );
37 g.DrawLine( pen, 90, 120, 110, 130 );
38 g.DrawLine( pen, 240, 30, 260, 40 );
39 g.DrawLine( pen, 240, 120, 260, 130 );
40
41 // draw top rectangle
42 g.DrawRectangle( pen, 110, 40, 150, 90 );
43
44 // set brush to red
45 brush.Color = Color.Red;
46
47 // draw base Ellipse
48 g.FillEllipse( brush, 280, 75, 100, 50 );
49
50 // draw connecting lines
51 g.DrawLine( pen, 380, 55, 380, 100 );
52 g.DrawLine( pen, 280, 55, 280, 100 );
53
54 // draw Ellipse outline
55 g.DrawEllipse( pen, 280, 30, 100, 50 );
56
57 } // end method OnPaint
58
59 } // end class LinesRectanglesOvals
```

Página 777 Deitel - C# How to Program

Animando uma sequência de imagens

O próximo exemplo faz a animação de uma série de imagens armazenadas em um vetor. A aplicação usa as técnicas para carregar e exibir **Images** daquelas ilustradas na Fig. 16.23. As imagens foram criadas com Adobe Photoshop.



Fig. 3 Animation of a series of images. (Part 1 of 3.)

A animação na figura 3 usa um **PictureBox**, que conterá as imagens que desejamos animar.

Usaremos um **Timer** para podermos circular entre as imagens, de forma que uma nova imagem seja exibida a cada 50 milissegundos por padrão.

A variável **count** mantém o rastreio do número da imagem atual e é incrementada cada vez que exibimos uma nova imagem.

O vetor inclui 30 imagens (numeradas de 0 a 29); quando a aplicação chega à imagem imagem 29, ela retorna para a imagem 0.

As 30 images foram preparadas previamente e colocadas na pasta **images** dentro da pasta **bin/Debug** do projeto.

```
1 // Fig. 16.24: LogoAnimator.cs
2 // Program that animates a series of images.
3
4 using System;
5 using System.Drawing;
6 using System.Collections;
7 using System.ComponentModel;
8 using System.Windows.Forms;
9 using System.Data;
10
11 // animates a series of 30 images
12 public class LogoAnimator : System.Windows.Forms.Form
13 {
14     private System.Windows.Forms.PictureBox logoPictureBox;
15     private System.Windows.Forms.Timer Timer;
16     private System.ComponentModel.IContainer components;
17
18     private ArrayList images = new ArrayList();
19     private int count = -1;
20
21     public LogoAnimator()           // form class constructor
22     {
23         InitializeComponent();
24
25         for ( int i = 0; i < 30; i++ )
26             images.Add( Image.FromFile( "images/deitel" + i +
27                               ".gif" ) );
28
29         // load first image
30         logoPictureBox.Image = ( Image ) images[ 0 ];
31
32         // set PictureBox to be the same size as Image
33         logoPictureBox.Size = logoPictureBox.Image.Size;
34
35     } // end constructor
36
37     [STAThread]
38     static void Main()
39     {
40         Application.Run( new LogoAnimator() );
41     }
42
43     // Visual Studio .NET generated code
44     private void Timer_Tick(object sender, System.EventArgs e )
45     {
46         // increment counter
47     }
48 }
```

```
47         count = ( count + 1 ) % 30;
48
49     // load next image
50     logoPictureBox.Image = ( Image )images[ count ];
51
52 } // end method Timer_Tick
53 } // end class LogoAnimator
```

Fig. 4 Animation of a series of images. (Part 2 of 3.)

Linhas 5 a 27 carregam cada uma das 30 imagens e as colocam em um **ArrayList**.

O método **Add** do **ArrayList** nos permite adicionar objetos ao **ArrayList**; usamos esse método nas linhas 26 e 27 para adicionar cada **Image**. Linha 30 coloca a primeira imagem no **PictureBox**, usando o indexador do **ArrayList**. Linha 33 modifica o tamanho do **PictureBox** de forma que ele fique igual ao tamanho da imagem que está exibindo.

O tratador de evento para o evento Tick do Timer (line 44 a 52) então exibe a próxima imagem a partir do **ArrayList**.

Dica de Desempenho 16.2

É mais eficiente carregar os quadros de animação como uma única imagem do que carregar cada uma separadamente. (Um programa de pintura, como Adobe Photoshop®, Jasc® or Paint Shop Pro™, podem ser usados para combinar os quadros da animação em uma única imagem, lado a lado ou formando uma matriz de imagens.) Se as imagens estão sendo carregadas separadamente pela Web, cada imagem carregada exigirá uma conexão separada ao site a partir onde elas estão armazenadas; este processo pode resultar em baixo desempenho.

Dica de Desempenho 16.3

A carga dos quadros de animação pode causar atrasos no programa, pois o programa esperaá que todas sejam carregadas antes de exibi-las.

6. C# e a Interface IComparable

Todas as classes C#, incluindo vetores são, em última análise, derivadas da classe base chamada `object`. A palavra reservada `object` é um apelido para a classe `System.Object`. O próximo fragmento de código identifica alguns dos métodos definidos na classe `System.Object`:

```
namespace System
{
    public class Object
    {
        public Object() { ... }
        public virtual bool Equals(object o) { ... }
        public virtual int GetHashCode() { ... }
        public Type GetType() { ... }
        public virtual string ToString() { ... }
        // ...
    }
}
```

Observe que a classe C# `System.Object` contém um método `Equals`, cujo propósito é indicar se algum outro objeto é “igual” a este. Por default, `obj1.Equals(obj2)` retorna `true` somente se `obj1` e `obj2` se referem ao mesmo objeto, ou seja, se apontam o mesmo espaço de memória instanciado para um objeto.

É claro que qualquer classe derivada pode sobrepor o método `Equals` para fazer a comparação da maneira mais apropriada a essa classe. Por exemplo, o método `Equals` é sobreposto na classe `System.Int32` como se segue:

Se `obj1` e `obj2` são `Int32`s, então
`obj1.equals(obj2)` é `true` quando `(int)obj1` é igual a `(int)obj2`.

Assim, todos os objetos C# fornecem uma maneira de testar a igualdade. Infelizmente, eles não fornecem uma maneira de testar se um objeto é “menor que” ou “maior que” outro. Para superar essa dificuldade, C# fornece a interface padrão chamada `IComparable`. O fragmento de código a seguir define a interface `IComparable`:

```
namespace System.Collections
{
    public interface IComparable
    {
        int CompareTo(object o);
    }
}
```

A interface `IComparable` define um único método. Este método de instância recebe um objeto “`o`” especificado como parâmetro e o compara com o objeto da instância que chama o método. O método retorna um inteiro que é menor que, igual a, ou maior que zero, dependendo se a instância deste objeto é menor que, igual a, ou maior que a instância do objeto “`o`” especificado como parâmetro.

Objetos Comparáveis Abstratos

Veja na página 49 que todos as classes definidas como uma estrutura de dados são derivadas da classe abstrata `ComparableObject`, que está no topo da hierarquia de classes. Como mostrado nessa figura, a classe `ComparableObject` implementa a interface `IComparable` discutida acima.

O método `CompareTo` é definido como um método abstrato no programa 6.1. Esse programa também define o método privativo `Compare`. Para entender a operação desse método, considere

uma expressão da forma `obj1.Compare(obj2)`. Primeiramente, o método `Compare` determina se `obj1` e `obj2` são instâncias do mesmo tipo (linha 7). Se assim for, o método `CompareTo` é chamado para fazer a comparação. Portanto, o método `CompareTo` sempre será invocado somente para instâncias da mesma classe.

```
1. public abstract class ComparableObject : IComparable
2. {
3.     public abstract int CompareTo(Object obj);
4.
5.     private int Compare(object obj)
6.     {
7.         if (GetType() == obj.GetType())
8.             return CompareTo(obj);
9.         else
10.             return GetType().FullName.CompareTo(
11.                 obj.GetType().FullName);
12.     }
13.
14.    public override bool Equals(object obj)
15.    {
16.        return Compare(obj) == 0;
17.    }
18.
19.    public override int GetHashCode()
20.    {
21.        return base.GetHashCode();
22.    }
23.
24.    public override string ToString()
25.    {
26.        return base.ToString();
27.    }
28. }
```

Programa 6.1: Métodos da classe ComparableObject

Se `obj1` e `obj2` são instâncias de tipos diferentes, então a comparação é baseada nos nomes dos tipos (linhas 10-11). Suponha que `obj1` é uma instância da classe chamada `Opus6.StackAsArray` e `obj2` é uma instância da classe chamada `Opus6.QueueAsLinkedList`. Então `obj1` é “menor que” `obj2` porque `StackAsArray` precede alfabeticamente `QueueAsLinkedList`.

Operadores de Comparação

O Programa 6.2 mostra como os operadores de comparação (`==`, `!=`, `<`, `<=`, `>` e `>=`) são implementados. Todos esses métodos invocam o método `Compare` e então interpretam o resultado da maneira adequada.

Os operadores `==` e `!=` são ligeiramente diferentes, para torná-los mais robustos e fáceis de usar. Especificamente, eles funcionam da maneira correta quando quaisquer dos operadorres de `==` e `!=` são referências null.

```
public abstract class ComparableObject : IComparable
{
    public static bool operator ==(ComparableObject c, object o)
    {
        if ((object)c == null || (object)o == null)
```

```
        return (object)c == (object)o;
    else
        return c.Compare(o) == 0;
}
public static bool operator !=(ComparableObject c, object o)
{
    if ((object)c == null || (object)o == null)
        return (object)c != (object)o;
    else
        return c.Compare(o) != 0;
}
public static bool operator <(ComparableObject c, object o)
{
    return c.Compare(o) < 0;
}
public static bool operator >(ComparableObject c, object o)
{
    return c.Compare(o) > 0;
}
public static bool operator <=(ComparableObject c, object o)
{
    return c.Compare(o) <= 0;
}
public static bool operator >=(ComparableObject c, object o)
{
    return c.Compare(o) >= 0;
}
//...
```

Programa 6.2: Operadores de comparação da classe ComparableObject

O uso de polimorfismo na forma acima fornece enorme poder ao programador. O fato de que todos os objetos são derivados da classe base ComparableObject, aliado ao fato de que cada classe concreta deverá implementar um método CompareTo apropriado, assegura que os operadores de comparação podem ser usados com quaisquer pares de objetos da classe ComparableObject e que as comparações sempre funcionarão como esperado.

Portanto, você pode declarar que cada estrutura de dados que implementaremos seja derivada de ComparableObject, de forma que essas estruturas tenham de implementar o método CompareTo. Ou, se preferir algo mais simples, pode declarar que cada estrutura é uma implementação da interface IComparable, e implementar e codificar o método CompareTo da forma mais adequada para essa estrutura.

Como exemplo, vejamos a classe Termo abaixo, que armazena um termo de um polinômio. Um polinômio, por sua vez, pode ser representado como uma lista ligada ordenada. A ordenação é dada pela sequência decrescente de expoentes dos termos que formam os nós da lista.

```
public class Termo : ComparableObject
{
    protected double coeficiente;
    protected int expoente;
    public Termo(double coef, int expo)
    {
        this.coeficiente = coef;
        this.expoente = expo;
```

```
    }
    public override int CompareTo(object obj)
    {
        Termo termo = (Termo)obj;
        if (expoente == termo.expoente)
        {
            if (coeficiente < termo.coeficiente)
                return -1;
            else
                if (coeficiente > termo.coeficiente)
                    return +1;
                else
                    return 0;
        }
        else
            return expoente - termo.expoente;
    }
    public void Diferenciar() // calcula derivada do termo
    {
        if (expoente > 0)
        {
            coeficiente *= expoente;
            expoente -= 1;
        }
        else
            coeficiente = 0;
    }
}
```

Em seguida, temos a classe Polinomio, que define o polinômio que agrupará os Termos.

```
abstract class Polinomio
{
    public abstract void Incluir(Termo termo); // incluirá Termo em ordem na lista
    public abstract void Diferenciar(); // percorrerá os termos e os diferenciará
    public abstract Polinomio SomadoA(Polinomio p); // devolve novo polinômio (this+p)
    public static Polinomio operator +(Polinomio p1, Polinomio p2)
    {
        return p1.SomadoA(p2);
    }
}
```

Procure implementar os métodos acima usando uma lista ligada para armazenar o polinômio e também os métodos da classe ComparableObject, que é a base de Termo, como vemos na classe a seguir:

```
using ListaLigada;
...
class PolinomioComoListaSimples : Polinomio
{
    ListaSimples<Termo> lista;
    public void Incluir(Termo termo) // incluirá Termo na lista, em ordem
decrescente de expoente
    {
    }
    public void Diferenciar() // percorrerá os termos e os diferenciará
    {
    }
}
```

```
public Polinomio SomadoA(Polinomio p) // devolve novo polinômio (this + p)
{
}
public Polinomio operator +(Polinomio p1, Polinomio p2)
{
    return p1.SomadoA(p2);
}
```

Constraints

Há momentos em que você deve assegurar que os elementos que você adiciona a uma lista genérica atendem certas restrições (por exemplo, que eles derivem de uma determinada classe base ou que eles implementem uma interface específica). No próximo exemplo, implementaremos uma lista ligada simples e ordenável. A lista consiste de NoListas, e cada NoLista deve garantir que os tipos adicionados a ela implementam IComparable. Isso é feito com o comando abaixo:

```
public class NoLista<T> : IComparable<NoLista<T>> where T : IComparable<T>
```

Isto define um NoLista genérico que armazena um valor do tipo T. NoLista de T implementa a interface IComparable<T>, o que significa que dois NoListas de T podem ser comparados. A classe NoLista é restrita (where T:IComparable<T>) a manter somente tipos que implementam a interface IComparable. Dessa maneira, você pode substituir T por qualquer tipo desde que esse tipo implemente IComparable.

O exemplo abaixo ilustra a implementação completa, com uma análise em seguida.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ListaLigadaComConstraints
{
    public class Funcionario : IComparable<Funcionario>
    {
        private string nome;
        public Funcionario(string nome)
        {
            this.nome = nome;
        }
        public override string ToString()
        {
            return this.nome;
        }

        // implement the interface
        public int CompareTo(Funcionario umFunc)
        {
            return this.nome.CompareTo(umFunc.nome);
        }
        public bool Equals(Funcionario outroFunc)
        {
            return this.nome == outroFunc.nome;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ListaLigadaComConstraints
{
    // NoLista deve implementar IComparable de NoLista de T.
    // restringe NoLista a somente aceitar itens que implementem IComparable
    // através do uso da palavra reservada where.

    public class NoLista<T> :
        IComparable<NoLista<T>> where T : IComparable<T>
    {
        // campos membros
        private T dados;
        private NoLista<T> proximo = null;
        private NoLista<T> anterior = null;

        // constructor
        public NoLista(T dados)
        {
            this.dados = dados;
        }

        // propriedades
        public T Dados { get { return this.dados; } }

        public NoLista<T> Prox
        {
            get { return this.proximo; }
        }

        public int CompareTo(NoLista<T> rhs)
        {
            // isto funciona devido à restrição
            return dados.CompareTo(rhs.dados);
        }

        public bool Equals(NoLista<T> rhs)
        {
            return this.dados.Equals(rhs.dados);
        }

        // métodos
        public NoLista<T> AdicionarNo(NoLista<T> novoNo)
        {
            if (this.CompareTo(novoNo) > 0) // deve entrar antes do atual
            {
                novoNo.proximo = this; // novo nó aponta para o atual

                // se temos um anterior, ele deve apontar o novo nó como
                // seu próximo
                if (this.anterior != null)
                {
                    this.anterior.proximo = novoNo;
                    novoNo.anterior = this.anterior;
                }
            }
        }
    }
}
```

```
// o campo anterior do nó atual aponta para o novoNo
this.anterior = novoNo;

// retorna o novoNo no caso em que ele seja o novo primeiro da lista
return novoNo;
}
else // deve entrar após o atual
{
    // se há um próximo ao atual, o novo nó é passado para
    // comparação
    if (this.proximo != null)
        this.proximo.AdicionarNo(novoNo);

    // Não temos um próximo, de forma que o novo nó será o próximo
    // do atual e o atual será o anterior do novo nó.
    else
    {
        this.proximo = novoNo;
        novoNo.anterior = this;
    }

    return this;
}
}

public override string ToString()
{
    string output = dados.ToString();

    if (proximo != null)
    {
        output += ", " + proximo.ToString();
    }

    return output;
}
} // end class
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ListaLigadaComConstraints
{
    public class ListaLigada<T> where T : IComparable<T>
    {
        // campos membros
        private NoLista<T> noCabeca = null;
        private int quantosNos = 0;

        // propriedades

        public int QuantosNos { get { return quantosNos; } }

        public NoLista<T> Primeiro { get { return noCabeca; } }
```

```

// indexador
public T this[int index]
{
    get
    {
        int cont = 0;
        NoLista<T> no = noCabeça;

        for (cont = 0; no != null && cont <= index; cont++)
        {
            if (cont == index)
                return no.Dados;
            no = no.Prox;
        } // end while
        throw new ArgumentOutOfRangeException();
    } // end get
} // end indexador

// construtor
public ListaLigada()
{
}

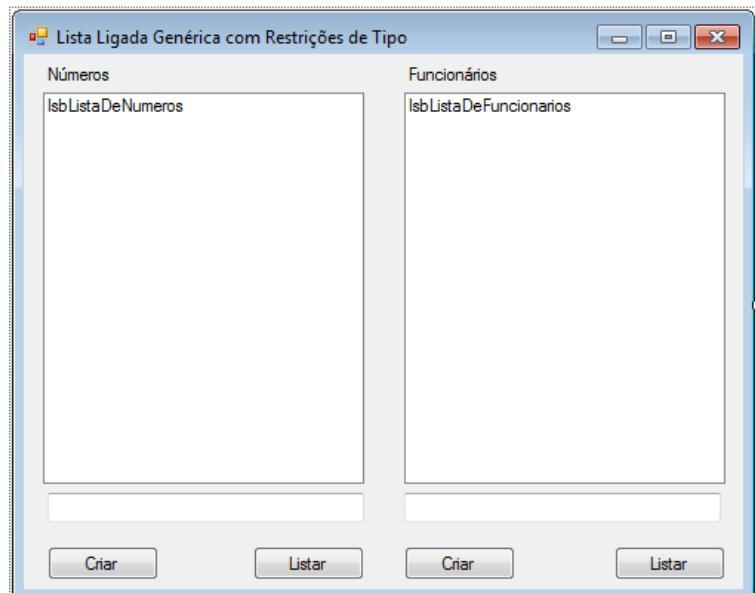
// métodos
public void AdicionarNaLista(T data)
{
    if (noCabeça == null)
        noCabeça = new NoLista<T>(data);
    else
        noCabeça = noCabeça.AdicionarNo(new NoLista<T>(data));
    quantosNos++;
}

public override string ToString()
{
    if (this.noCabeça != null)
        return this.noCabeça.ToString();
    else
        return string.Empty;
}
}

// Formulário de Teste
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace ListaLigadaComConstraints
{
    public partial class frmLista :
Form
    {
        ListaLigada<int> minhaLista;
        ListaLigada<Funcionario>
funcionarios;
        public frmLista()
        {
            InitializeComponent();
        }
    }
}

```



```
}

private void brnCriarListaDeNumeros_Click(object sender, EventArgs e)
{
    minhaLista = new ListaLigada<int>();
    Random rand = new Random();
    for (int i = 0; i < 10; i++)
    {
        int nextInt = rand.Next(10);
        minhaLista.AdicionarNaLista(nextInt);
    }
}

private void btnCriarListaDefuncionarios_Click(object sender, EventArgs e)
{
    funcionarios= new ListaLigada<Funcionario>();
    funcionarios.AdicionarNaLista(new Funcionario("João"));
    funcionarios.AdicionarNaLista(new Funcionario("Paulo"));
    funcionarios.AdicionarNaLista(new Funcionario("Jorge"));
    funcionarios.AdicionarNaLista(new Funcionario("Ringo"));
}

private void btnListarNumeros_Click(object sender, EventArgs e)
{
    lsbListaDeNumeros.Items.Clear();
    for (int indice = 0; indice < minhaLista.QuantosNos; indice++)
        lsbListaDeNumeros.Items. txtNumeros.Text = minhaLista.ToString();
}

private void btnListarFuncionarios_Click(object sender, EventArgs e)
{
    lsbListaDefuncionarios.Items.Clear();
    NoLista<Funcionario> noAtual = funcionarios.Primeiro;
    while (noAtual != null)
    {
        lsbListaDefuncionarios.Items.Add(noAtual.Dados);
        noAtual = noAtual.Prox;
    }

    txtFuncionarios.Text = funcionarios.ToString();
}
}
}
```

Neste exemplo, começamos declarando uma classe que pode ser colocada na lista ligada:

```
public class Funcionario : IComparable<Funcionario>
```

Esta declaração indica que objetos Funcionario são comparáveis, e observamos que a classe Funcionario implementa os métodos obrigatórios (CompareTo e Equals). Observe que estes métodos são seguros em relação ao tipo de dados, pois eles sabem que o parâmetro a eles passado será do tipo Funcionario). A classe ListaLigada, por sua vez, é declarada para armazenar apenas tipos que implementem IComparable:

```
public class ListaLigada<T> where T : IComparable<T>
```

de forma que estamos garantidos em relação à capacidade de ordenar a lista. ListaLigada armazena objetos do tipo NoLista. NoLista também implementa IComparable e exige que os objetos que ela armazena também implementem IComparable:

```
public class NoLista<T> : IComparable<NoLista<T>> where T : IComparable<T>
```

Estas restrições tornam seguro e simples implementar o método CompareTo de NoLista porque NoLista sabe que estará comparando com outros NoListas cujos dados são comparáveis:

```
public int CompareTo(NoLista<T> rhs)
{
    // isto funciona devido à restrição declarada acima
    return data.CompareTo(rhs.data);
}
```

Note que não tivemos de testar rhs para verificar se ele implementa IComparable; nós já tínhamos restringido NoLista para que ele armazene apenas dados que implementam IComparable.

7. Listas Ligadas revisitadas

A lista ligada simples é a mais básica de todas as estruturas de dados que usam alocação dinâmica.

Ela é, simplesmente, uma sequência de objetos alocados dinamicamente, cada um dos quais indica (referencia) seu sucessor na lista.

Mesmo tendo uma óbvia simplicidade, existem inúmeras variações de implementação. A figura 6.1 mostra diversas das mais comuns variações de listas ligadas.

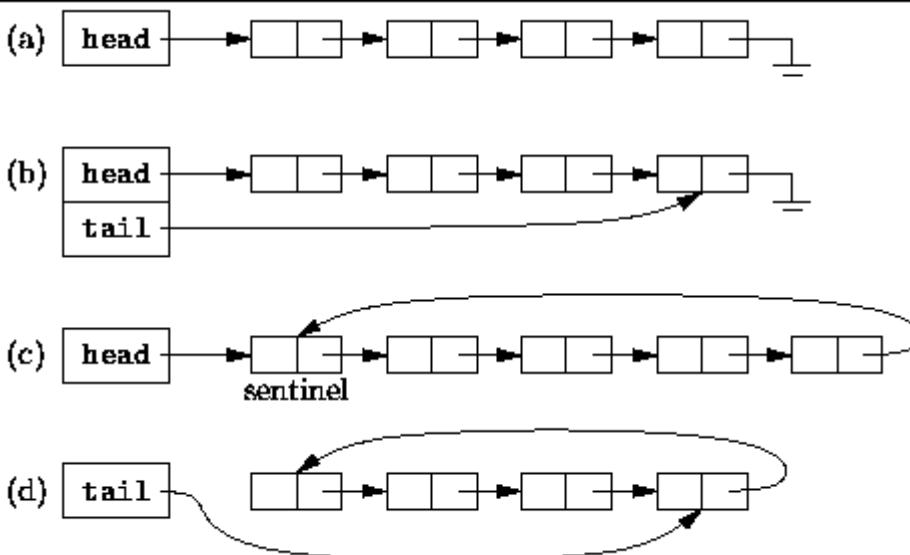


Figura 6.1: Variações de lista ligada simples

A lista ligada simples mais básica é mostrada na Figura 6.1. (a). Cada elemento da lista aponta (ou referencia) o seu sucesso e o último elemento contém a referência null. Uma variável, chamada `head` na Figura 6.1 (a), é usada para manter o rastreio da lista a partir de seu início.

A lista ligada simples da Figura 6.1. (a) é ineficiente naqueles casos em que desejamos adicionar elementos nas duas pontas da lista. Enquanto é fácil adicionar elementos no início da lista, para adicioná-los na outra extremidade (a cauda, `tail`) precisaremos localizar o último elemento. Se esse modelo de lista é usado, precisaremos percorrer toda a lista para encontrar seu último elemento.

A Figura 6.1 (b) mostra uma implementação da lista ligada simples que torna mais eficiente a adição de elementos após o final da lista. Essa solução usa uma segunda variável, `tail`, que referencia o último elemento da lista. Obviamente, essa eficiência de tempo vem pelo custo do espaço adicional usado para armazenar a variável `tail`.

Já a lista ligada simples identificada por (c) na Figura 6.1 ilustra dois “truques” comuns de programação. Existe um elemento extra no início da lista chamado `sentinela`. Este nó nunca é usado para manter dados mas está sempre presente na lista. A vantagem principal de usar um nó

sentinela é que ele simplifica a programação de certas operações. Por exemplo, já que sempre há um nó sentinela “mantendo guarda”, nunca precisaremos modificar a variável `head`. De forma semelhante, com esse nó sentinela nunca teremos de temer que a lista fique vazia e que essa situação possa comprometer a execução do programa ao gerar exceções. No entanto, o nó sentinela traz a desvantagem de ocupar mais memória e de ter de ser criado quando a lista é instanciada (geralmente, no método construtor da lista).

A lista (c) também é uma *lista circular*. Ao invés de usarmos uma referência `null` para demarcar o final da lista, o último elemento da lista referencia o nó sentinela. A vantagem desta abordagem é que a inserção de novos elementos antes do início da lista, a inserção ao final da lista e a inserção em uma posição arbitrária qualquer da lista passam a ser operações idênticas.

Obviamente também é possível criar uma lista ligada circular que não use um nó sentinela, e isso é exibido na Figure 6.1 (d), que mostra uma variação na qual uma única variável é usada para manter o rastreio da lista. Observe que, nessa figura, a variável, `tail`, referencia o último elemento da lista. Já que a lista é circular neste caso, o primeiro elemento segue o último nó da lista. Portanto, é relativamente simples inserir novos nós tanto antes do início quanto após o final da lista. Esta variação diminui o gasto de memória extra, ao custo de um pouco mais de tempo de processamento em certas operações.

A Figura 6.2 ilustra como se representa a lista vazia (isto é, a lista que não contém nenhum nó) , para cada uma das variações dadas na Figure 6.1. Note que o nó sentinela sempre está presente na variação de lista (c). Por outro lado, nas variações de lista simples que não usam um nó sentinela, a referência `null` é usada para indicar uma lista vazia.

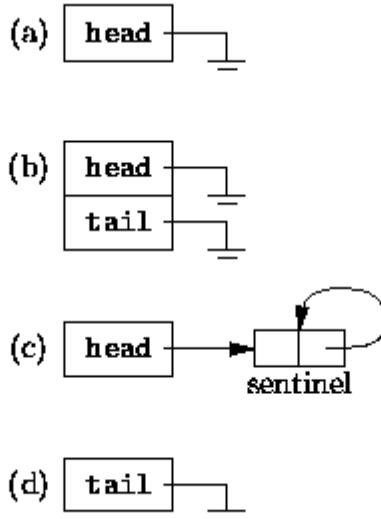


Figura 6.2: Listas ligadas simples vazias.

Nas próximas seções, apresentaremos os detalhes de implementação de uma lista ligada simples genérica. Escolhemos apresentar a variação (b) – aquela que usa os campos `head` e `tail` – já que ela suporta eficientemente as operações de antepor (inclusão antes do primeiro nó) e anexação (inclusão após o último nó).

Uma Implementação

A Figura 6.3 ilustra o esquema de lista simplesmente ligada que escolhemos implementar.

Duas estruturas relacionadas são usadas.

Os elementos da lista são representados usando instâncias da classe `NoLista` que contém 3 campos, `lista`, `dados` e `prox`. A estrutura principal é uma instância da classe `ListaSimples` que também contém 2 campos apontadores, `primeiro` e `ultimo`, que referenciam, respectivamente, o primeiro e último nós da lista e o campo inteiro `quantosNos`, que indica o número de elementos armazenados na lista a cada instante.

O campo `lista` de cada `NoLista` contém uma referência à instância de `ListaSimples` ao qual o nó está associado. O campo `dados` é usado para referenciar objetos armazenados em cada nó da lista e o campo `prox` referencia o próximo nó da lista.

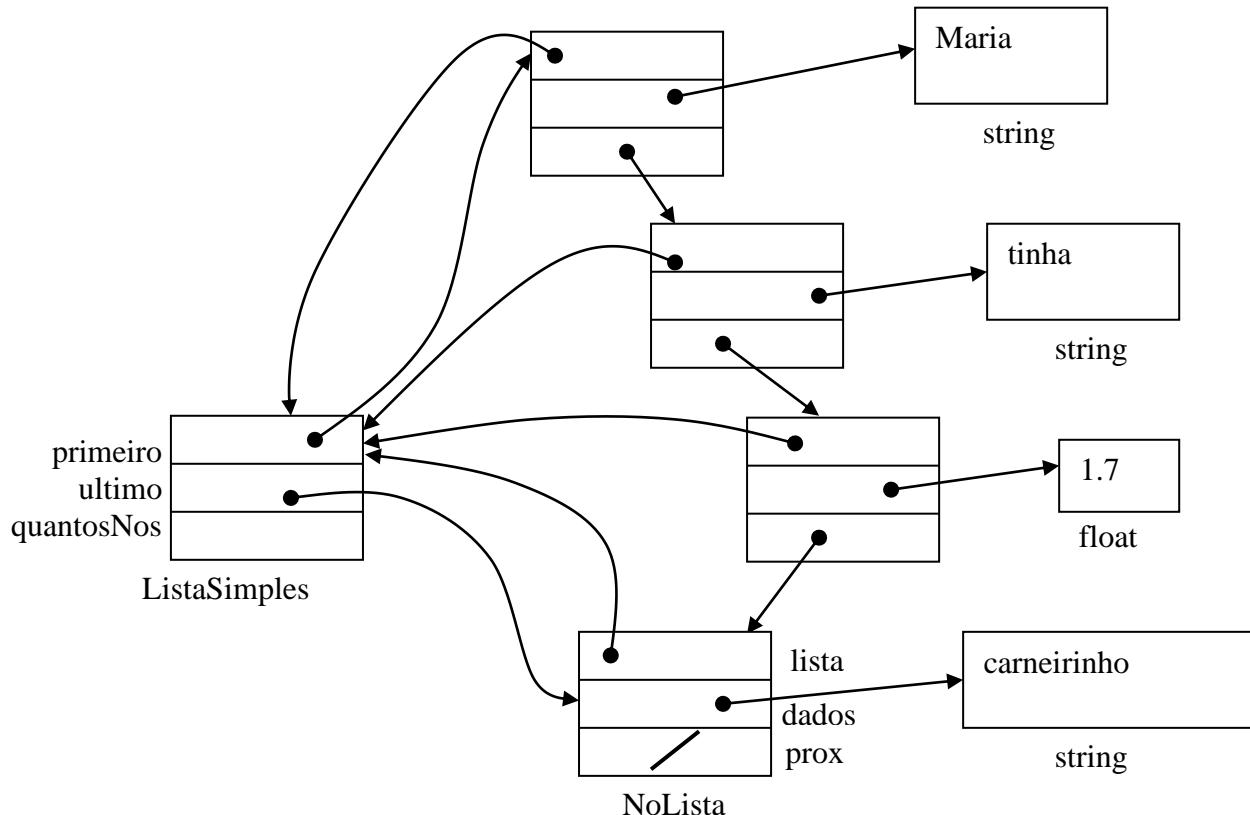


Figura 6.3: Representação na memória de uma lista ligada simples.

O programa 6.1 define a classe `ListaSimples.NoLista`. Ela é usada para representar os nós de uma lista ligada. Ela tem três campos, `lista`, `dados` e `prox`, um construtor e duas propriedades, `Dados` e `Prox`. Programa 6.1 também define os campos da lista ligada `ListaSimples`, `primeiro`, `ultimo`, e `quantosNos`.

```

public class ListaSimples<Tipo>
{
    protected NoLista<Tipo> primeiro;
    protected NoLista<Tipo> ultimo;
    protected int quantosNos;

    public sealed class NoLista<Tipo>
    {
        internal ListaSimples<Tipo> lista;
        internal Tipo dados;
        internal NoLista<Tipo> prox;

        internal NoLista(ListaSimples<Tipo> lista, Tipo dados,
                        NoLista<Tipo> prox)
        {
            this.lista = lista;
        }
    }
}
  
```

```
        this.dados = dados;
        this.prox = prox;
    }

    public Tipo Dados
    {
        get { return dados; }
        set { dados = value; }
    }

    public NoLista<Tipo> Prox
    {
        get { return prox; }
        set { prox = value; }
    }
}
```

Programa 6.1: Classe ListaSimples.NoLista.

Elementos da Lista

As definições dos métodos da classe ListaSimples.NoLista são dadas no Programa 6.1. Conjuntamente, existem três métodos – um construtor e duas propriedades.

O construtor simplesmente inicializa os campos com os valores fornecidos. Assinalando valores para os campos `lista`, `dados` e `prox` toma uma quantidade constante de tempo. Portanto, o tempo de execução do construtor é $O(1)$.

As propriedades `Dados` e `Prox` fornecem métodos acessadores `get` e `set` que simplesmente retornam e atribuem os valores dos campos correspondentes. Claramente, o tempo de execução desses métodos é $O(1)$.

Construtor Default de ListaSimples

O código para o construtor default da classe ListaSimples é dado no Programa 6.2. Como os campos `primeiro` e `ultimo` são, inicialmente, `null`, a lista é vazia por default. Como resultado disso, o construtor não faz nada de especial. O tempo de execução desse construtor default é claramente uma constante. Ou seja, $T(n)=O(1)$.

```
public ListaSimples()
{
}
```

Programa 6.2: Construtor da Classe ListaSimples

Método Limpar

O Programa 6.3 fornece o código para o método `Limpar()` da classe ListaSimples. O objetivo desse método é descartar os conteúdos atuais da lista e torná-la novamente vazia. Claramente, o tempo de execução de `Limpar()` é $O(1)$.

```
protected NoLista primeiro;
protected NoLista ultimo;
protected int quantosNos;

public void Limpar()
{
    primeiro = null;
    ultimo = null;
```

```
        quantosNos = 0;  
    }
```

Programa 6.3: Método Limpar() da Classe ListaSimples

```
using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace ListaLigada  
{  
    class ListaSimples<Tipo>  
    {  
        protected NoLista<Tipo> primeiro;  
        protected NoLista<Tipo> ultimo;  
        protected int quantosNos;  
  
        public NoLista<Tipo> Primeiro // propriedade para ponteiro do inicio da lista  
        {  
            get { return primeiro; }  
        }  
  
        public NoLista<Tipo> Ultimo // propriedade para ponteiro do final da lista  
        {  
            get { return ultimo; }  
        }  
  
        public bool EstaVazia  
        {  
            get { return primeiro == null; }  
        }  
  
        public Tipo PrimeiroValor // acessa o valor armazenado no 1º nó da lista  
        {  
            get  
            {  
                if (EstaVazia) // lista vazia  
                    throw new ListaVaziaException();  
                return primeiro.dados;  
            }  
        }  
  
        public Tipo UltimoValor // acessa o valor armazenado no 1º nó da lista  
        {  
            get  
            {  
                if (ultimo == null) // lista vazia ou não tem apontador para o último nó  
                    throw new ListaVaziaException();  
                return ultimo.dados;  
            }  
        }  
    }
```

```
public void IncluirAntesDoPrimeiro(Tipo item) // passa o dado como parâmetro
{
    // cria um novo nó para armazenar o dado (item) e o faz apontar o
    // 1º nó atual da lista
    NoLista<Tipo> tmp = new NoLista<Tipo> (this, item, primeiro);
    if (EstaVazia)
        ultimo = tmp;
    primeiro = tmp;
    quantosNos++;
}

public void IncluirAposUltimo(Tipo item)
{
    // cria um novo nó para armazenar o dado (item) e esse nó não
    // aponta para nenhum nó seguinte (pois será o último da lista)
    NoLista<Tipo> tmp = new NoLista<Tipo> (this, item, null);

    if (EstaVazia) // se a lista está vazia, o primeiro nó apontará o novo criado
        primeiro = tmp;
    else           // se a lista não está vazia, o atual último nó se liga ao novo
        ultimo.prox = tmp;

    ultimo = tmp; // o novo nó passa a ser o último da lista
    quantosNos++;
}

public void Copiar(ListaSimples<Tipo> origem, bool manterDestino)
{
    if (origem != this)
    {
        if (!manterDestino) // se manterAtual = false, apaga a lista de destino
            Limpar(); // limpa a lista de destino (this) para receber os nós da origem

        for (NoLista ptr = origem.primeiro; ptr != null; ptr = ptr.prox)
            IncluirAposUltimo(ptr.dados);
    }
}

public void Extrair(Tipo item)
{
    NoLista<Tipo> atual = primeiro;
    NoLista<Tipo> anterior = null;
    while (atual != null && !atual.dados.Equals(item))
    {
        anterior = atual;
        atual = atual.prox;
    }
    if (atual == null)
        throw new ArgumentException("Item não encontrado para extração");
```

```
if (atual == primeiro)
    primeiro = primeiro.prox;
else
    anterior.prox = atual.prox; // se atual for o último nó, o prox do anterior
                                // será null
if (atual == ultimo)
    ultimo = anterior;
quantosNos--;
}

public ListaSimples()
{
}

public void Limpar()
{
    primeiro = null;
    ultimo = null;
    quantosNos = 0;
}

public sealed class NoLista
{
    internal ListaSimples<Tipo> lista;
    internal Tipo dados;
    internal NoLista prox;

    public void InsereDepois(Tipo item) // inseré novo nó após o nó que invoca
                                         // este método
    {
        prox = new NoLista<Tipo> (lista, item, prox);
        lista.quantosNos++;
        if (lista.ultimo == this)
            lista.ultimo = prox;
    }

    public void InsereAntes(Tipo item) // insere novo nó antes do nó que invoca
                                         // este método
    {
        NoLista<Tipo> tmp = new NoLista<Tipo> (lista, item, this); // novo nó é criado e aponta
                                                                     // para o nó atual
        lista.quantosNos++;
        // se o nó que invocou este método (atual) é o primeiro da lsita,
        // o novo nó passará a ser o primeiro
        if (this == lista.primeiro)
            lista.primeiro = tmp;
        else // se o nó atual não é o primeiro, procura-se o nó anterior ao atual
        {
            // para ligá-lo ao novo nó, que já está ligado ao atual
            NoLista<Tipo> percorre = lista.primeiro;
            while (percorre != null && percorre.prox != this)
```

```
    percorre = percorre.prox;
    percorre.prox = tmp;
}
}

internal NoLista(ListaSimples<Tipo> lista, Tipo dados, NoLista prox)
{
    this.lista = lista;
    this.dados = dados;
    this.prox = prox;
}

public Tipo Dados
{
    get { return dados; }
    set { dados = value; }
}

public NoLista Prox
{
    get { return prox; }
    set { prox = value; }
}
}
}
}
```

8. Algums Estruturas em Delphi e Java

A classe Fila em Delphi

```
unit uFila;

interface

Const MAXIMO = 500;      // tamanho máximo do vetor que implementa a fila

Type

  TFila<Dado> = class
    public
      function tamanho():integer;           // retorna número de posições em uso
      function estaVazia():boolean;        // devolve true se fila está vazia
      function oInicio():Dado;
      function oFim():Dado;
      procedure enfileirar(o: Dado);
      function retirar() : Dado;
      constructor iniciar(MaximoAtual : Integer);
    private
      area: array[1..Maximo] of Dado;      // vetor de objetos usado como
                                            // área de armazenamento
      posicoes,                         // número de elementos armazenados Maximo
      inicio,                            // índice do início da fila
      fim : integer;                     // índice do fim da fila
    end;

implementation

uses Dialogs;

function TFila.tamanho():integer;   // retorna número de posições em uso
begin
  tamanho := (posicoes - inicio + fim) mod posicoes;
end;

function TFila.estaVazia():boolean; // devolve true se fila está vazia,
begin
  estaVazia := inicio = fim;       // e false caso contrário
end;

function TFila.oInicio():Dado;
Var o : Dado;
begin
  if estaVazia then
    Raise Exception.Create('Underflow da fila');

  oInicio := area[inicio]; // devolve o objeto inicial
                           // sem retirá-lo da fila
end;

function TFila.oFim():Dado;
var o : Dado;
begin
  if estaVazia() then
    Raise Exception.Create('Underflow da fila');
```

```
if fim = 0 then
  o := area[posicoes-1] // devolve o objeto do final da fila
else
  // sem retirá-lo da fila
  o := area[fim-1];
oFim := o;
end;

procedure TFila.enfileirar(o: Dado);
begin
  if tamanho() = posicoes-1 then
    Raise Exception.Create('Overflow de Fila');

  area[fim] := o; // inclui elemento na primeira posição
  fim := (fim + 1) mod posicoes; // calcula próxima posição livre
end;

function TFila.retirar() : Dado;
var o : Dado;
begin
  if estaVazia() then
    Raise Exception.Create('Underflow da fila');

  o := F[inicio]; // copia o elemento inicial da fila
  inicio = (inicio +1) mod posicoes; // calcula novo inicio da fila
  retirar := o; // devolve elemento inicial
end;

Constructor TFila.Iniciar(MaximoAtual : Integer);
Begin
  Posicoes := MaximoAtual;
  Inicio := 0;
  Fim := 0;
End;

end.
```

A CLASSE PILHA EM DELPHI

```
Unit uPilhaVetor;
Interface
uses SysUtils;
const
  Max_Pilha = 100;
type
  TPilha = class // (TObject)
    Public
      procedure empilhar(elemento : TObject); // Estas rotinas são acessíveis
      function desempilhar : TObject;
      function tamanho : integer;
      function estaVazia : boolean;
      function oTopo : TObject;
      constructor iniciar; // chamada automaticamente quando se cria a pilha
    private
      area : Array [1..Max_Pilha] of TObject; // Estes campos são inacessíveis
      Topo : 0..Max_Pilha; // externamente, pois são PRIVATE
      procedure transbordou;
      procedure esvaziou;
    end;
implementation
```

Note que a classe acima deriva de TObject diretamente, que o vetor AREA é um vetor de TObject. Isso indica que nesse vetor pode-se armazenar qualquer tipo de objeto, pois todo objeto é, em última análise, um TObject (herança).

Mas nenhum tipo simples, primitivo, como Integer, char, boolean, Record, array são descendentes de TObject. Um TObject é como uma “casca” que envolve campos e rotinas, e os tipos primitivos não possuem essa “casca”. Por isso, são incompatíveis (type mismatch). Para guardar um valor primitivo no vetor acima, é NECESSÁRIO que esse valor seja ENGLOBADO por um objeto. Por exemplo, para guardar um valor char num objeto, fazemos algo como:

```
{ OS EXEMPLOS ABAIXO SÃO DA APLICAÇÃO QUE USA A CLASSE TPILHA }
Tchar = class
  Caracter : char;
End;
```

No entanto, para guardar um caracter (lido do teclado, por exemplo) no vetor da classe Tpilha, precisamos ENGLOBAR (encapsular) esse caracter na classe Tchar:

```
Var   umCaracter : Tchar;
      CaracterLido : char;           // vai ser lido do teclado ou de arquivo
      UmaPilha : Tpilha;
Begin
  ...
  umaPilha := Tpilha.create;
  ...
  readln(caracterLido);
  UmCaracter := Tchar.create; // cria o objeto da classe Tchar
  UmCaracter.caracter := caracterLido; // encapsula o valor lido
```

Todo objeto, mesmo o da classe Tchar, é um TObject. Assim, passa a ser compatível com o parâmetro da rotina Empilhar, que armazena os dados no vetor AREA. AREA é private, de modo que não se pode acessá-lo diretamente. Exemplo:

umaPilha.area[umaPilha.topo] := umCaracter; → por AREA ser private, não pode fazer isto!

umaPilha.empilhar(umCaracter); → isto é permitido, pois empilhar() é Public

UmCaracter é um descendente de TObject, por isso, como a “assinatura” de empilhar exige um TObject, essa passagem de parâmetro é aceita

Para recuperar o valor que foi armazenado no vetor AREA, é NECESSÁRIO que se CONVERTA o que está armazenado para o tipo do OBJETO que receberá o valor desejado. Armazenamos um Tchar, mas internamente a AREA, este foi convertido para TObject. Agora, para recuperá-lo, temos de fazer essa conversão, usando o operador **AS**, que utiliza um objeto como se fosse uma classe desejada:

<Objeto> **as** <classe desejada>

Exemplo:

```
Var caracterAntesEmpilhado : Tchar;
...
CaracterAntesEmpilhado := umaPilha.desempilhar AS Tchar;
```

Desempilhar retorna um TObject

Converte o TObject para a classe TChar

IMPLEMENTAÇÃO DOS MÉTODOS DE UMA CLASSE

Método é o nome que se dá para procedimentos e funções que são descritos dentro de uma classe.

São também chamados de FUNÇÕES-MEMBRO, pois estão dentro da classe.

Na parte **Implementation** de uma Unit em Delphi, escrevemos os códigos dos procedimentos e funções membros da classe (ou seja, os métodos). Também podemos escrever procedimentos e funções privativos, que não são chamados por outras units. Esses métodos privativos podem ser chamados, quando necessário, pelos métodos públicos.

Exemplo:

```
procedure TPilha.empilhar(elemento : TObject);
begin
  if topo = Max_Pilha then
    transbordou()
  else
    begin
      inc(topo);
      area[topo] := elemento;
    end;
end;

function TPilha.desempilhar : TObject;
var elemento : TObject;
begin
```

Recebe um TObject e o armazena no vetor AREA. Se o vetor estiver cheio, chama Transbordou, que é um método privativo. Na chamada, qualquer **objeto** pode ser passado como parâmetro

Devolve um objeto genérico (TObject) que, após ser recebido no local de chamada, deve ser convertido para o tipo de classe específico do objeto que anteriormente foi armazenado. Para isso, usa-se o operador de CAST (conversão) **AS**

```

if estaVazia() then
begin
    desempilhar := nil;
    esvaziou()
end
else
    begin
        elemento := area[topo];
        dec(topo);
        desempilhar := elemento;
    end;
end;

```

USO DA CLASSE TPILHA

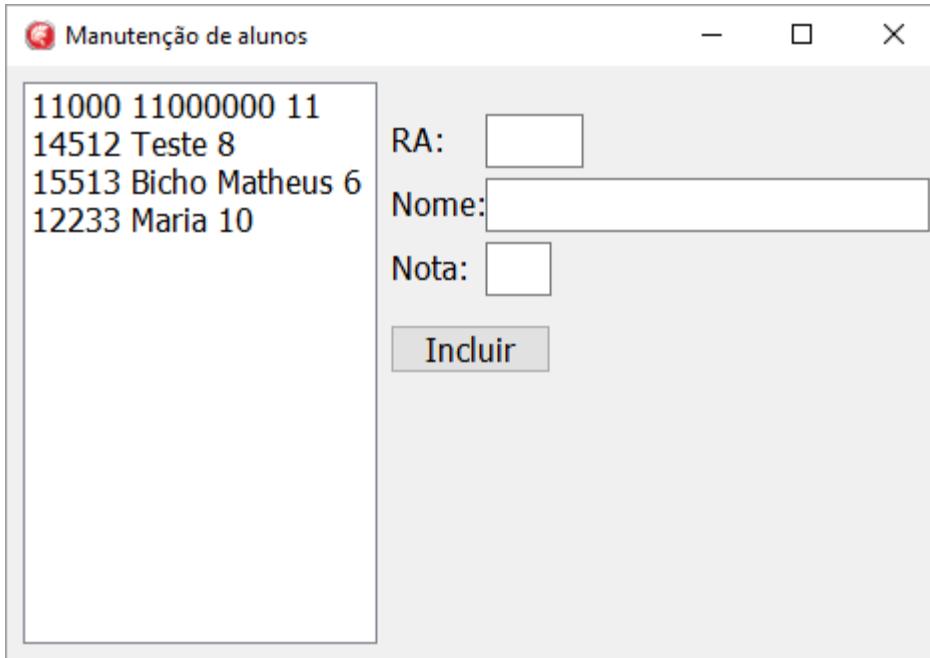
```

procedure TfrmAnalisa.btnCloseAnalisaClick(Sender: TObject);
var
    p           : TPilha; // declara a pilha usada para analisar o balanceamento
    Balanceada : Boolean; // informa se a seqüência é balanceada ou não
    indice      : integer; // usado para percorrer a string com a cadeia a analisar
    caracterLido : char; // caractere atualmente sob análise, copiado da string
    Abertura,
    caracterDesempilhado : TSimbolo; // objetos para encapsular os caracteres
                                      // analisados
    Entrada          : String;
begin
    Entrada       := edExpressao.text;
// abaixo a pilha é instanciada, ou seja, é criada e passa a existir na memória
    p             := TPilha.create; // Inicializa a PILHA, que fica VAZIA
    Balanceada   := True;
    indice        := 1;
    While (indice <= length(Entrada)) and Balanceada do
    Begin
        caracterLido := Entrada[indice];
        If caracterLido In ['{','[','('] Then
            begin
                // Cria objeto para encapsular (guardar) o caractere lido, pois a pilha só
                // aceita Objetos
                Abertura := TSimbolo.create;
                Abertura.simbolo := caracterLido;
                // empilha o objeto que encapsulou o caractere
                p.empilhar(Abertura)
            end
        Else
            If caracterLido In ['}',']','')'] Then
                If p.estaVazia Then
                    Balanceada := False { pois a pilha já esvaziou }
                Else
                    begin
                        { cria objeto para guardar o caractere desempilhado }
                        caracterDesempilhado := TSimbolo.create;
                        caracterDesempilhado := p.desempilhar as TSimbolo;
                        If Not combinam(caracterLido, caracterDesempilhado.simbolo) Then
                            Balanceada := False;
                    end;
                    inc(indice);
                end;
            If not p.estaVazia Then
                Balanceada := False;
            chkBalanceada.checked := Balanceada;
    end;

```

Lista Ligada implementada em Delphi

Abaixo temos o código de um programa simples, feito em Delphi, que utiliza um arquivo de registros de Alunos para leitura e armazenamento em uma lista ligada. O formulário da figura abaixo representa a interface com o usuário desse programa.



A interface abaixo foi criada para facilitar a comparação entre Strings.

```
unit uInterfaceString;

interface

type
  IString = interface
    function toString:string;
  end;

implementation

end.
```

A seguir temos a implementação das classes NoLista e ListaSimples em Delphi:

```
TListaSimples<T : IComparable<T>, IString> = class
  primeiro,
  ultimo,
  atual,
  anterior : TNoLista<T>;
  quantosNos : integer;
  private
    primeiroAcessoDoPercorso : boolean;
  public
    constructor create;
    function estaVazia : boolean;
    procedure insereAntesDoInicio(novoNo : TNoLista<T>);
    procedure insereAposFim(novoNo : TNoLista<T>);
    procedure insereAposNo(qualNo, novoNo : TNoLista<T>);
    procedure insereEmOrdem(novoNo : TNoLista<T>);
    procedure removerNo(qualNo, noAnterior : TNoLista<T>); overload;
    procedure removerNo; overload;
    procedure listar(listbox : TListBox);
    procedure iniciarPercorsoSequencial;
    function podePercorrer : boolean;
    function existeEmOrdem(procurado : T) : boolean;
  end;

implementation

{ TNoLista }

constructor TNoLista<T>.create(informacao: T; proximo: TNoLista<T>);
begin
  info := informacao;
  prox := proximo;
end;

{ TListaSimples }

constructor TListaSimples<T>.create;
begin
  primeiro := nil;
  ultimo := nil;
  atual := nil;
  anterior := nil;
  quantosNos := 0
end;

function TListaSimples<T>.estaVazia: boolean;
begin
  estaVazia := primeiro = nil;
  // result := primeiro = nil;
  // ou
  // if primeiro = nil then
  //   estaVazia := true
  // else
  //   estaVazia := false;
end;

function TListaSimples<T>.existeEmOrdem(procurado: T): boolean;
```

```

var
  achouMaior,
  achouIgual : boolean;
begin

  achouIgual := false;
  achouMaior := false;
  anterior   := nil;
  atual      := primeiro;
  while (not achouIgual) and (not achouMaior) and
    (atual <> nil) do
  begin
    if atual.Info.compareTo(procurado) = 0 then
      achouIgual := true
    else
      if atual.Info.compareTo(procurado) > 0 then
        achouMaior := true
      else
        begin
          anterior := atual;
          atual    := atual.prox;
        end;
    end;
  end;
  existeEmOrdem := achouIgual;
end;

procedure TListaSimples<T>.iniciarPercorsoSequencial;
begin
  atual           := primeiro;
  primeiroAcessoDoPercorso := true;
end;

procedure TListaSimples<T>.insereAntesDoInicio(
  novoNo: TNoLista<T>);
begin
  if estaVazia then
    ultimo     := novoNo;

  novoNo.prox := primeiro;
  primeiro   := novoNo;
  inc(quantosNos);
end;

procedure TListaSimples<T>.insereAposFim(novoNo: TNoLista<T>);
begin
  if estaVazia then
    primeiro := novoNo
  else
    ultimo.prox := novoNo;

  novoNo.prox := nil;
  ultimo     := novoNo;
  inc(quantosNos);
end;

procedure TListaSimples<T>.insereAposNo(qualNo, novoNo: TNoLista<T>);
begin
  if (qualNo = nil) or estaVazia then
    raise Exception.Create('Local inválido para inserção');

  novoNo.prox := qualNo.prox;
  qualNo.prox := novoNo;

```

```
inc(quantosNos);

if qualNo = ultimo then
    ultimo := novoNo;
end;

procedure TListaSimples<T>.insereEmOrdem(novoNo: TNoLista<T>);
begin
    if estaVazia then
        insereAntesDoInicio(novoNo)
    else
        if anterior = ultimo then
            insereAposFim(novoNo)
        else
            if anterior = nil then
                insereAntesDoInicio(novoNo)
            else
                begin
                    novoNo.prox := atual;
                    anterior.prox := novoNo;
                    inc(quantosNos);
                end;
            end;
    end;

procedure TListaSimples<T>.listar(listbox: TListBox);
begin
    listbox.Clear;
    atual := primeiro;
    while atual <> nil do
    begin
        listbox.items.add(atual.Info.ToString());
        atual := atual.prox;
    end;
end;

function TListaSimples<T>.podePercorrer: boolean;
begin
    if atual = nil then
        result := false
    else
        if primeiroAcessoDoPercorso then
            begin
                primeiroAcessoDoPercorso := false;
                result := true
            end
        else
            begin
                atual := atual.prox;
                result := atual <> nil;
            end;
    end;

procedure TListaSimples<T>.removerNo(qualNo, noAnterior: TNoLista<T>);
begin
    if not estaVazia then
    begin
        if qualNo = primeiro then
            begin
                primeiro := primeiro.prox;
                if primeiro = nil then // só havia um único nó
                    ultimo := nil;
            end;
    end;
```

```

end
else
  if qualNo = ultimo then
    begin
      noAnterior.prox := nil;
      ultimo          := noAnterior;
    end
  else
    begin
      noAnterior.prox := qualNo.prox;
      qualNo.prox     := nil;
    end;

  qualNo.destroy;
  dec(quantosNos);
end;
end;

procedure TListaSimples<T>.removerNo;
begin
  removerNo(atual, anterior);
end;

end.

```

Logo abaixo temos a classe que representa a informação a ser armazenada, ou seja, dados de alunos:

```

unit uClasseAluno;

interface

uses uInterfaceString, sysUtils;

type
  RAluno = record
    ra   : string[5];
    nome : string[30];
    nota : real;
  end;

// TAluno = class;

TAluno = class(TInterfacedObject, IComparable<TAluno>, IString)
  registro : RAluno;

  constructor create(novoRegistro : RAluno);

  function CompareTo(outroAluno : TAluno):integer; overload;
  function compareTo(T : TObject) : integer; overload;
  function ToString : string;
end;

implementation

{ TAluno }

function TAluno.compareTo(outroAluno : TAluno): integer;
begin
  if registro.ra < outroAluno.registro.RA then
    result := -1
  else
    result := 1;
end;

```

```

else
  if registro.ra = outroALuno.registro.RA then
    result := 0
  else
    result := 1;
end;

function TAluno.CompareTo(T: TObject): integer;
begin
  result := 0;
end;

constructor TAluno.create(novoRegistro: RAluno);
begin
  registro := novoRegistro;
end;

function TAluno.ToString : string;
begin
  result := registro.RA+' '+registro.Nome+' '+
            FloatToStr(registro.nota);
end;

end.

```

O código abaixo é o arquivo de codificação dos componentes do formulário e seus eventos. Como exercício implemente as funcionalidades para gravar os dados no arquivo de registros ao final da execução do programa, para excluir um aluno da lista, para procurar e exibir os dados de um aluno cujo RA seja digitado.

```

unit uFormAluno;

interface

uses
  Winapi.Windows,      Winapi.Messages,       System.SysUtils,       System.Variants,
System.Classes, Vcl.Graphics,
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls;

type
  TfrmAluno = class(TForm)
    lsbAlunos: TListBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    edRA: TEdit;
    edNome: TEdit;
    edNota: TEdit;
    btnIncluir: TButton;
    procedure FormShow(Sender: TObject);
    procedure btnIncluirClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmAluno: TfrmAluno;

```

```
implementation

{$R *.dfm}

uses uClasseAluno, uListaSimples;

var
  Aluno    : RAluno; // registro para ler e escrever num arquivo
  oAluno   : TAluno; // objeto da classe TAluno
  arquivo  : file of RAluno;

  lista    : TListaSimples<TAluno>;
  novoNo  : TNoLista<TAluno>;

procedure TfrmAluno.btnAddClick(Sender: TObject);
begin
  with aluno do
  begin
    // preenchemos o registro com os dados do formulário
    while length(RA) < 5 do RA := '0'+RA;

    while length(nome) < 30 do nome := nome+' ';

    RA := edRA.Text;
    nome := edNome.Text;
    nota := strToFloat(edNota.Text);

    // encapsulamos (boxing) desse registro num objeto
    // da classe TAluno:
    oAluno := TAluno.create(aluno);

    // existeEmOrdem ajusta os valores dos apontadores
    // anterior e atual para que indiquem o local de
    // inclusão ou de exclusão ou de acesso
    if lista.existeEmOrdem(oAluno) then
      ShowMessage('Já existe aluno com esse RA')
    else
    begin
      // criamos um nó de lista para armazenar o objeto
      // que encapsula o registro com os dados digitados
      novoNo := TNoLista<TAluno>.create(oAluno, nil);

      // incluimos esse registro no local da lista
      // que foi determinado na pesquisa
      // (existeEmOrdem)
      lista.insereEmOrdem(novoNo);

      lista.listar(lsbAlunos); // exibe resultado
    end;
  end;
end;

procedure TfrmAluno.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  rewrite(arquivo);
  lista.iniciarPercorsoSequencial;
  while lista.podePercorrer do // esta função anda na lista
  begin
    oAluno := lista.atual.Info;
    write(arquivo, oAluno.registro); // unboxing
  end;
  closeFile(arquivo);
end;
```

```
end;

procedure TfrmAluno.FormShow(Sender: TObject);
begin
  lista := TListaSimples<TAluno>.create;

  assignFile(arquivo, 'alunos.dat');
  if not fileExists('Alunos.dat') then
    rewrite(arquivo)
  else
  begin
    reset(arquivo);
    while not eof(arquivo) do
    begin
      // lemos um registro do arquivo
      read(arquivo, aluno); // aluno é um registro RAluno

      // não podemos armazenar um registro específico na
      // nossa lista e, por isso, criamos um objecto
      // compatível que encapsule esse registro
      oAluno := TAluno.Create(aluno); // encapsula aluno lido
                                      // num objeto TAluno

      // criamos nó da lista contendo o objeto TAluno
      // com os dados do aluno lido; esse nó de lista
      // é genérico e, por isso, usamos <TAluno> para
      // informarmos ao construtor que deve armazenar
      // um TAluno
      novoNo := TNoLista<TAluno>.create(oAluno, nil);

      lista.insereAposFim(novoNo);
    end;
  end;
  closeFile(arquivo);
  lista.listar(lsbAlunos);
end;

end.
```

A Classe FilaVetor em Java

Abaixo temos uma implementação em linguagem Java, para mostrar as diferenças em relação a C#. Observe que apenas os trechos com diferenças foram enfatizados, de forma que os demais comandos são iguais nas duas linguagens.

```
public class FilaVetor implements Queue {
    public static final int MAXIMO = 500; // tamanho default do vetor F
    ...
    public FilaVetor() // construtor que utiliza o default MAXIMO
    {
        this(MAXIMO); // chama o método construtor com parâmetro
    } // utilizando o polimorfismo do construtor
    public Object oInicio() throws FilaVaziaException {
        ...
    }
    public Object oFim() throws FilaVaziaException {
        ...
    }
    public void enfileirar(Object o) throws FilaCheiaException {
        ...
    }
    public Object retirar() throws FilaVaziaException {
        ...
    }
}
```

Implementação de PilhaVetor em Java:

```
public interface Stack<T> {
    public void empilhar(T elemento); // empilha objeto da classe T
    public T desempilhar() // devolve objeto do topo,
    throws PilhaVaziaException; // retira-o da pilha e dispara
    // exceção caso pilha vazia
    public T oTopo() // devolve objeto do topo
    throws PilhaVaziaException; // sem retirá-lo da pilha e
    // dispara exceção caso vazia
    public int tamanho(); // devolve número de elementos da pilha
    public boolean estaVazia(); // informa se pilha está vazia ou não
}

public class PilhaVaziaException extends RuntimeException {
    public PilhaVaziaException (String erro) {
        super(erro); // chama o método da classe ancestral
    }
}

public class PilhaVetor<Dado> implements Stack<Dado> {

    public static final int MAXIMO = 500; // tamanho default do vetor P
    private int posicoes; // tamanho dado pela aplicação
    private Dado P[]; // vetor de Dado, com tamanho genérico,
    // usado como área de armazenamento
    private int topo = -1; // índice do topo da pilha
    public PilhaVetor() { // construtor que utiliza o default MAXIMO
        this(MAXIMO); // chama o método construtor com parâmetro
    }
}
```

```
// utilizando o polimorfismo do construtor
}

public PilhaVetor(int posic) { // construtor polimórfico
    posicoes = posic; // armazena o tamanho do vetor
    P = new Object[posicoes]; // P é um vetor de objetos; new cria um
} // vetor com o tamanho indicado e o
// associa a P

public int tamanho() { // devolve o número de posições em uso
    return (topo + 1);
}

public boolean estaVazia() { // devolve true se pilha está vazia,
    return (topo < 0); // e false caso contrário
}

public void empilhar(Dado o) throws PilhaCheiaException {
    if (tamanho() == posicoes)
        throw new PilhaCheiaException("Overflow de Pilha");
    P[++topo] = o; // soma 1 ao topo e armazena objeto nessa posição
}

public Dado oTopo() throws PilhaVaziaException {
    Object o;
    if (estaVazia())
        throw new PilhaVaziaException("Underflow da pilha");
    o = P[topo]; // devolve o objeto do topo mas
    return o; // não decremente o topo
}

public Dado desempilhar() throws PilhaVaziaException {
    Object o;
    if (estaVazia())
        throw new PilhaVaziaException("Underflow da pilha");
    o = P[topo]; // copia o elemento do topo
    P[topo] = null; // libera memória
    topo--; // decremente o topo
    return o; // devolve elemento que estava no topo
}

public class PilhaCheiaException extends RuntimeException {
    public PilhaCheiaException (String erro) {
        super(erro);
    }
}
```

Implementação com Lista Ligada em Java

```
public class PilhaLista implements Stack {
    private NoLista topo;
    private int tamanho;

    public PilhaLista () { // construtor
        topo = null;
        tamanho = 0;
    }

    public int tamanho() {
        return tamanho;
    }

    public boolean estaVazia() {
        return (topo == null);
```

```
}

public void empilhar(Object o) {
    NoLista novoNo = new NoLista();
    novoNo.setElemento(o); // coloca elemento no nó
    novoNo.setProximo(topo); // liga novo nó ao antigo topo da pilha
    topo = novoNo; // topo passa a apontar o novo nó
    tamanho++; // atualiza número de elementos na pilha
}

public Object oTopo() throws PilhaVaziaException {
    Object o;
    if (estaVazia())
        throw new PilhaVaziaException("Underflow da pilha");
    o = topo.getElemento();
    return o;
}

public Object desempilhar() throws PilhaVaziaException {
    if (estaVazia())
        throw new PilhaVaziaException("Underflow da pilha");
    Object o = topo.getElemento(); // obtém o objeto do topo
    topo = topo.getProximo(); // avança topo para o nó seguinte
    tamanho--; // atualiza número de elementos na pilha
    return o; // devolve o objeto que estava no topo
}
}
```

Usando Herança de ListaSimples

```
public class PilhaLista implements Stack extends ListaSimples {

    public PilhaLista () { // construtor
        super();
    }

    public void empilhar(Object o) {
        NoLista novoNo = new NoLista(o,null);
        InsereNoInicio(novoNo);
    }

    public Object oTopo() throws PilhaVaziaException {
        Object o;
        if (estaVazia())
            throw new PilhaVaziaException("Underflow da pilha");
        o = getPrimeiro().getElemento();
        return o;
    }

    public Object desempilhar() throws PilhaVaziaException {
        if (estaVazia())
            throw new PilhaVaziaException("Underflow da pilha");
        Object o = removePrimeiro(); // avança topo para o nó seguinte
        return o; // devolve o objeto que estava no topo
    }
}
```

Na classe ListaSimples, criaremos o método RemoverPrimeiro(), que removerá o primeiro elemento da lista ligada e atualizará os demais ponteiros corretamente.

```
Public NoLista RemoverPrimeiro()
```

```
{  
    If (EstaVazia)  
        throws new ListaVaziaException("lista vazia");  
    NoLista devolvido = Primeiro; // primeiro é um NoLista  
    Primeiro = Primeiro.Prox; // avança o ponteiro para o 2º nó  
    if (Primeiro == null) // se, ao remover o 1º nó, a lista ficou vazia, então  
        Ultimo = null; // atualizamos o ponteiro ultimo para não apontar nenhum nó  
        tamanho--;  
    return devolvido;  
}  
}
```

Na classe `ListaSimples`, criamos o método `removePrimeiro()`, que removerá o primeiro elemento da lista ligada e atualizará os demais ponteiros corretamente.

```
public NoLista removePrimeiro()  
{  
    If (estaVazia())  
        throws new ListaVaziaException("lista vazia");  
    NoLista devolvido = getPrimeiro(); // primeiro é um NoLista  
    setPrimeiro (getPrimeiro().getProximo()); // avança o ponteiro para o 2º nó  
    if (primeiro == null) // se, ao remover o 1º nó, a lista ficou vazia, então  
        ultimo = null; // atualizamos o ponteiro ultimo para não apontar nenhum nó  
        tamanho--;  
    return devolvido;  
}
```

Aplicativo em Java usando arquivos de acesso direto e árvores binárias de busca

O arquivo será um arquivo normal, ou seja, uma sequência de bytes. Os campos deverão ter tamanho fixo, como acontece num arquivo de registros criado com Delphi ou C. Portanto, é interessante criarmos uma classe abstrata genérica para todos os tipos de registro e que obrigue o implementador a criar métodos para abrir arquivo, ler registros, escrever registros, posicionar no registro desejado, etc. Essa é a classe abstrata `Dados<T>`, que é genérica e, portanto, pode ser aplicada a qualquer tipo de registro que desejarmos.

`RandomAccessFile` é um tipo de arquivo de Java que permite posicionar a leitura ou a escrita em um byte específico. Assim, conhecendo o registro que desejamos acessar, basta calcular o número de seu byte inicial no arquivo, para podermos posicionar o acesso a partir desse byte.

O método `lerRegistro` tem como parâmetros o arquivo que se deseja ler e a posição desse arquivo que será lida, além do registro já instanciado que receberá os dados lidos.

O método `gravarRegistro` possui como parâmetros o arquivo onde se deseja gravar os dados, a posição nesse arquivo onde será feita a gravação e o registro já instanciado que contém os dados a serem gravados.

O método `escreveString` será usada para escrever cadeias de caracteres com tamanho fixo no arquivo.

```
import java.io.IOException;  
import java.io.RandomAccessFile;  
  
public abstract class Dados<T>  
{  
    public abstract void lerRegistro(RandomAccessFile f, long posicao, T dados)  
        throws IOException;  
    public abstract void gravarRegistro(RandomAccessFile f, long posicao, T dados)  
        throws IOException;  
    public abstract void escreveString(RandomAccessFile f, String s, int tamanho)  
        throws IOException;
```

}

Como desejamos ler dados de alunos, devemos criar uma classe Aluno, que conterá os campos desse registro. Essa classe, que consideraremos como uma "Classe de Dados", bem como todas as outras classes que desejarmos usar para armazenar dados em forma de registro, deverão definir exatamente o tamanho de cada campo, pois esse tamanho influenciará o cálculo do byte inicial de cada registro dentro do arquivo, bem como influirá na leitura e escrita de cada campo. A definição do tamanho de cada campo é feita por meio de valores inteiros estáticos e finais, fazendo com que apenas uma cópia desses valores exista para todas as instâncias da classe de dados que estamos criando, e impossibilitando a mudança de seus valores em classes derivadas.

É importante que essa classe implemente a interface Comparable<Aluno>, para que se possa comparar registros de alunos e, também, garantir que a compilação apenas permita a comparação de alunos entre si, não entre um aluno e outro objeto qualquer.

O uso do modificador de visibilidade protected garante que esses campos possam ser usados nas classes derivadas da classe de dados mas, mesmo assim, não sejam acessíveis diretamente no programa.

```
public class Aluno implements Comparable<Aluno>
{
    protected final static int tamanhoRA      = 5;
    protected final static int tamanhoNome    = 30;
    protected final static int tamanhoCurso   = 2;
    private final static int tamanhoPeriodo = 4;
    protected static final int tamanhoRegistro = tamanhoRA + tamanhoNome + tamanhoCurso + tamanhoPeriodo;

    protected String RA;                  // máximo de 5 posições
    protected String nome;                // máximo de 30 posições
    protected String curso;               // máximo de 2 posições
    protected int periodo;                // 4 bytes, segundo o padrão de java

    public Aluno() {
        this("", "", "", 0);
    }

    public Aluno(String r, String n, String c, int p) {
        setRA(r);
        setNome(n);
        setCurso(c);
        setPeriodo(p);
    }

    public int compareTo(Aluno outroAluno) {
        return getRA().compareTo(outroAluno.getRA());
    }

    public String getCurso() {
        return curso;
    }

    public void setCurso(String curso) {
        while (curso.length()<2)
            curso = "0"+curso;
        this.curso = curso;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {      // garante que nome possua 30
        caracteres
        if (nome.length() > 30)           // trunca caracteres além da 30a posição
    }
}
```

```
        nome = nome.substring(0, 30);
while (nome.length() < 30 )
    nome = nome+" ";
this.nome = nome;
}

public int getPeriodo() {
    return periodo;
}

public void setPeriodo(int periodo) {
    this.periodo = periodo;
}

public String getRA() {
    return RA;
}

public void setRA(String ra) { // garante que o RA possua 5 dígitos
    if (ra.length() > 5) // trunca caracteres além da 5a posição
        ra = ra.substring(0, 5);
    while (ra.length() < 5)
        ra = "0"+ra;
    RA = ra;
}

public String toString() {
    return RA;
}
}
```

A classe a seguir é derivada de Dados<Aluno>, de forma que é obrigada a implementar os métodos abstratos de Dados<T>, usando os valores de um objeto da classe Aluno. Essa classe, portanto, implementa as funcionalidades necessárias para o funcionamento da leitura e escrita de dados em um arquivo de alunos.

Se tivéssemos outro tipo de registro, além de criar a classe de dados para esse outro tipo de registro, deveríamos também criar uma classe Registro para esse tipo, como abaixo. Ela garante que a leitura e a escrita levaram em conta os tamanhos de cada campo do registro em questão, além de posicionar a leitura e a escrita nos bytes iniciais dos registros desejados.

```
import java.io.*;
public class RegistroAluno extends Dados<Aluno> {

    public void lerRegistro(RandomAccessFile f, long posicao, Aluno dados)
        throws IOException
    {
        char nomeLido[] = new char[dados.tamanhoNome];
        char raLido[] = new char[dados.tamanhoRA];
        char cursoLido[] = new char[dados.tamanhoCurso];
        int periodoLido,
            indice;

        f.seek(posicao * dados.tamanhoRegistro);
        for (indice=0; indice < dados.tamanhoRA; indice++)
            raLido[indice] = f.readChar();
        for (indice=0; indice < dados.tamanhoNome; indice++)
            nomeLido[indice] = f.readChar();
        for (indice=0; indice < dados.tamanhoCurso; indice++)
            cursoLido[indice] = f.readChar();
        periodoLido = f.readInt();
        dados.setRA(new String(raLido).replace('\0', ' '));
        dados.setNome(new String(nomeLido).replace('\0', ' '));
    }
}
```

```
        dados.setCurso(new String(cursoLido).replace('\0', ' '));
        dados.setPeriodo(periodoLido);
    }

public void gravarRegistro(RandomAccessFile f, long posicao, Aluno dados)
    throws IOException
{
    try {
        f.seek(posicao * dados.tamanhoRegistro);
    }
    finally
    {
        escreveString(f, dados.getRA(), dados.tamanhoRA);
        escreveString(f, dados.getNome(), dados.tamanhoNome);
        escreveString(f, dados.getCurso(), dados.tamanhoCurso);
        f.writeInt(dados.getPeriodo());
    }
}

public void escreveString(RandomAccessFile f, String s, int tamanho)
    throws IOException
{
    StringBuffer cadeia = null;
    if (s != null)
        cadeia = new StringBuffer(s);
    else
        cadeia = new StringBuffer(tamanho);
    cadeia.setLength(tamanho);
    f.writeChars(cadeia.toString());
}
}
```

Abaixo temos uma nova versão da classe NoArvore, que foi adaptada a este estudo. Note que ela implementa o método compareTo, ao invés de getChave devolvendo String. Assim, fica mais genérica:

```
public class NoArvore2<T extends Comparable<T>>
{
    T elemento;
    private NoArvore2 esq, dir;

    public NoArvore2 (T e, NoArvore2 esq, NoArvore2 dir) {
        setElemento(e);
        setEsq(esq);
        setDir(dir);
    }

    public NoArvore2 getDir() {
        return dir;
    }

    public int compareTo(T outroNo) {
        return elemento.compareTo(outroNo);
    }

    public void setDir(NoArvore2 dir) {
        this.dir = dir;
    }

    public NoArvore2 getEsq() {
        return esq;
    }
}
```

```
public void setEsq(NoArvore2 esq) {
    this.esq = esq;
}

public T getElemento() {
    return elemento;
}

public void setElemento (T e) {
    elemento = e;
}
```

A classe Arvore também foi modificada, para aceitar nós genéricos e permitir o uso da interface Comparable nas comparações que, ao invés de exigirem um método getChave() no nó, passam a aceitar a comparação feita pelo próprio objeto armazenado no nó.

```
public class Arvore2<T extends Comparable<T>> {
    private NoArvore2<T> raiz, atual, antecessor;
    private int tamanho;

    public Arvore2(){
        raiz = null;
        antecessor = null;
        atual = null;
        tamanho = 0;
    }

    public NoArvore2<T> getAntecessor() {
        return antecessor;
    }

    public void setAntecessor(NoArvore2<T> antecessor) {
        this.antecessor = antecessor;
    }

    public NoArvore2<T> getAtual() {
        return atual;
    }

    public void setAtual(NoArvore2<T> atual) {
        this.atual = atual;
    }

    public NoArvore2<T> getRaiz() {
        return raiz;
    }

    public void setRaiz(NoArvore2<T> raiz) {
        this.raiz = raiz;
    }

    public int getTamanho() {
        return tamanho;
    }

    public void setTamanho(int tamanho) {
        this.tamanho = tamanho;
    }

    public boolean existe(T dados) {
        antecessor = null;
        atual = raiz;
```

```
while (atual != null)
{
    if (atual.getElemento().compareTo(dados) == 0)
        return true;
    else
    {
        antecessor = atual;
        if (atual.getElemento().compareTo(dados) > 0 )
            atual = atual.getEsq();
        else
            atual = atual.getDir();
    }
}
return false;
}

public boolean incluiNo(T dados)
{
    if (existe(dados))
        return false;
    else
    {
        NoArvore2<T> novo = new NoArvore2<T>(dados, null, null);
        if (raiz == null)
            raiz = novo;
        else
            if (antecessor.getElemento().compareTo(dados) > 0 ) // antecessor
maior
                antecessor.setEsq(novo);
            else
                antecessor.setDir(novo);

        this.setTamanho(this.getTamanho()+1);
        return true;
    }
}

private void liga(NoArvore2<T> pai, T novo, NoArvore2<T> neto)
{
    if (pai.getElemento().compareTo(novo)>0)
        pai.setEsq(neto);
    else
        pai.setDir(neto);
}

public boolean excluiNo(T excluido)
{
    if (!existe(excluido))
        return false;
    else
    {
        // antecessor e atual foram definidos em existe()
        if (atual.getDir() == null) // nó a excluir tem 0 ou 1 filho?
            liga(antecessor, excluido, atual.getEsq());
        else
            if (atual.getEsq() == null)
                liga(antecessor, excluido, atual.getDir());
            else
            {
                // nó a excluir tem 2 filhos
                antecessor = atual;
                NoArvore2<T> aux = atual.getEsq();
```

```
        while (aux.getDir() != null) // procura maior dos menores
filhos
{
    antecessor = aux;
    aux = aux.getDir();
}

atual.setElemento(aux.getElemento()); // troca conteúdo
antecessor.setDir(aux.getEsq());
aux = null;
}
this.setTamanho(this.getTamanho()-1);
return true;
}
}
```

E, por fim, temos um programa com interface gráfica que nos permite ver a árvore e os RAs de cada nó. Você deve implementar os tratamentos dos botões e exibir os resultados no Frame onde a árvore é apresentada.

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MantemAlunos extends JFrame
{
    private static final long serialVersionUID = 1L;
    private JButton btnIncluir, btnExcluir, btnBuscar, btnSair;
    private JPanel pnlEdicao; // container dos campos de dados
    private JTextField edRA, edNome, edCurso, edPeriodo;
    private JLabel lblRA, lblNome, lblCurso, lblPeriodo;
    static private JInternalFrame frame;
    static private MeuJPanel pnlDesenho;

    private static Arvore2<Aluno> arv = new Arvore2<Aluno>();
    private static RandomAccessFile arqAlunos;
    private static long posicaoDeGravacao;

    public MantemAlunos() // construtor de Editor que criará o JFrame, colocará seu
    {                               // título, estabelecerá um tamanho para o formulário e o
        // exibirá
        super("Manutenção de Alunos com Árvore de Busca"); // cria JFrame; põe
        setTitle("Manutenção de Alunos com Árvore de Busca");
        setSize(1024, 768); // dimensões do formulário em pixels
        btnIncluir = new JButton("Incluir", new ImageIcon("Add.bmp"));
        btnExcluir = new JButton("Excluir", new ImageIcon("Minus.bmp"));
        btnBuscar = new JButton("Buscar", new ImageIcon("Oeil2.bmp"));
        btnSair = new JButton("Sair", new ImageIcon("Close1.bmp"));

        btnIncluir.setSize(100, 30);
        btnExcluir.setSize(100, 30);
        btnBuscar.setSize(100, 30);
        btnSair.setSize(100, 30);

        // cria os controles de edição
        lblRA = new JLabel("RA:");
        edRA = new JTextField();
```

```
edRA.setSize(50, 24);
lblNome = new JLabel("Nome:");
edNome = new JTextField();
edNome.setSize(240, 24);
lblCurso = new JLabel("Curso:");
edCurso = new JTextField();
edCurso.setSize(20,24);
lblPeriodo = new JLabel("Período:");
edPeriodo = new JTextField();
edPeriodo.setSize(10,24);

// cria um JPanel para colocar os visores e o label acima:
pnlEdicao = new JPanel();

// informa que o layout do pnlVisores será BorderLayout
pnlEdicao.setLayout(new GridLayout(1,12,3,3));

// Adicionamos os controles de edição em sequencia

pnlEdicao.add(lblRA);
pnlEdicao.add(edRA);
pnlEdicao.add(lblNome);
pnlEdicao.add(edNome);
pnlEdicao.add(lblCurso);
pnlEdicao.add(edCurso);
pnlEdicao.add(lblPeriodo);
pnlEdicao.add(edPeriodo);
pnlEdicao.add(btnIncluir);
pnlEdicao.add(btnExcluir);
pnlEdicao.add(btnBuscar);
pnlEdicao.add(btnSair);

Container cntForm = getContentPane(); // acessa o painel de conteúdo do frame
cntForm.setLayout(new BorderLayout());
cntForm.add(pnlEdicao, BorderLayout.NORTH);

JDesktopPane panDesenho = new JDesktopPane();
cntForm.add(panDesenho);

frame = new JInternalFrame("Dados da árvore", true, true, true, true);
panDesenho.add(frame);
frame.setSize(this.getWidth()-20, this.getHeight()-80);
frame.show();
frame.setOpaque(true);

pnlDesenho = new MeuJPanel();
Container cntFrame = frame.getContentPane();
cntFrame.add(pnlDesenho);

setVisible(true); // exibe o formulário
}

public static RandomAccessFile abrirArquivoAlunos()
{
    JFileChooser dlgAbrir = new JFileChooser();
    dlgAbrir.setCurrentDirectory(new File("."));
    dlgAbrir.setFileSelectionMode(JFileChooser.FILES_ONLY);

    int result = dlgAbrir.showOpenDialog(null);
    if (result == JFileChooser.CANCEL_OPTION)
        return null;
    File arquivo = dlgAbrir.getSelectedFile();

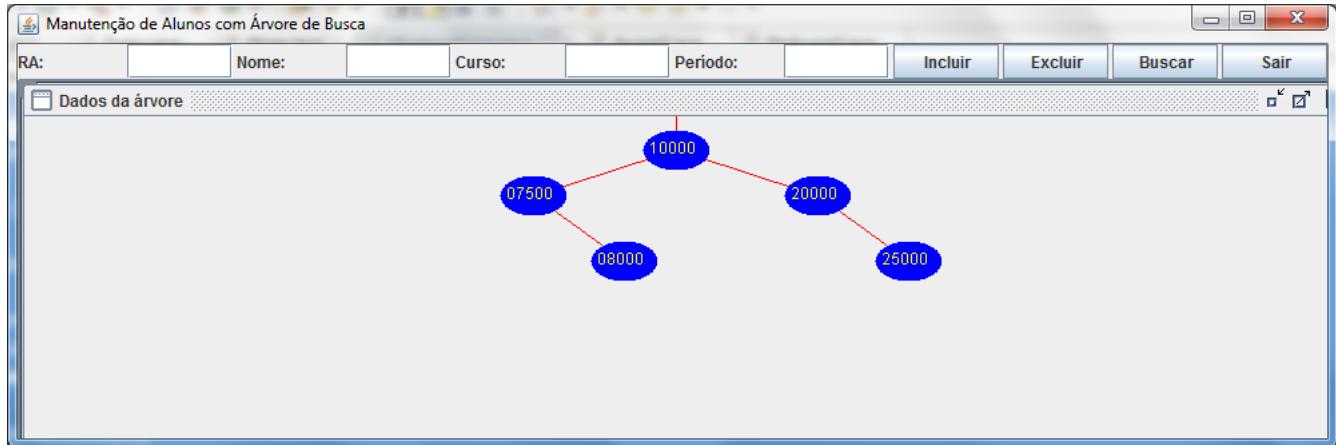
    if (arquivo == null || arquivo.getName().equals(""))
    {
```

```
JOptionPane.showMessageDialog(null, "Nome de arquivo inválido",
                           "Arquivo inválido",
JOptionPane.ERROR_MESSAGE);
    return null;
}
else
{
    try
    {
        RandomAccessFile arquivoAberto = new
RandomAccessFile(arquivo, "rw");
        return arquivoAberto;
    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(null, "Arquivo não existe",
                           "Nome inválido de arquivo",
JOptionPane.ERROR_MESSAGE);
        return null;
    }
}
/*
    File arquivo = new File("alunos.dat");
try {
    RandomAccessFile arquivoAberto = new
RandomAccessFile(arquivo, "rw");
    return arquivoAberto;
}
catch (IOException e)
{
    JOptionPane.showMessageDialog(null, "Arquivo não existe", "Nome
inválido de arquivo", JOptionPane.ERROR_MESSAGE);
    return null;
}
*/
}

public static void leituraDaArvore(long inicio, long fim) throws IOException
{
    if (inicio <= fim)
    {
        int meio = (int) (inicio + fim) / 2;
        RegistroAluno regAluno = new RegistroAluno();
        Aluno umAluno = new Aluno();
        regAluno.lerRegistro(arqAlunos, meio, umAluno);
        arv.incluiNo(umAluno);
        leituraDaArvore(inicio, meio-1);
        leituraDaArvore(meio+1,fim);
    }
}

public static void incluirDados()
{
    String novoRA;
    do
    {
        novoRA = JOptionPane.showInputDialog("RA:");
        if (!novoRA.equals(""))
        {
            String novoNome = JOptionPane.showInputDialog("Nome:");
            String novoCurso= JOptionPane.showInputDialog("Curso:");
            int novoPeriodo = Integer.parseInt(
```

```
JOptionPane.showInputDialog("Periodo:");  
    Aluno umAluno = new Aluno(novoRA, novoNome, novoCurso, novoPeriodo);  
    if (!arv.incluiNo(umAluno))  
        System.out.println("esse aluno já existe");  
    else  
        System.out.println("Aluno incluido");  
    }  
}  
while (!novoRA.equals(""));  
}  
  
public static void gravaArvore(NoArvore2<Aluno> atual, RandomAccessFile f)  
throws IOException {  
    if (atual != null)  
    {  
        gravaArvore(atual.getEsq(), f);  
        Aluno umAluno = (Aluno) atual.getElemento();  
        RegistroAluno regAluno = new RegistroAluno();  
        regAluno.gravarRegistro(f, posicaoDeGravacao++, umAluno);  
        gravaArvore(atual.getDir(), f);  
    }  
}  
  
public static void salvarDados() throws IOException  
{  
    arqAlunos.close();  
    arqAlunos = new RandomAccessFile(new File("alunos.dat"), "rw");  
    posicaoDeGravacao = 0;  
    gravaArvore(arv.getRaiz(), arqAlunos);  
    arqAlunos.close();  
}  
  
public static void exibeOpcoes() throws IOException  
{  
    int opcao;  
    do  
    {  
        opcao = Integer.parseInt(JOptionPane.showInputDialog(  
                "1 - Incluir\n2- Listar \n3 - sair\nSua  
opção:"));  
        switch (opcao)  
        {  
            case 1 : incluirDados();  
            case 2 : arv.inOrdem(arv.getRaiz());  
            case 3 : salvarDados();  
        }  
    } while (opcao != 3);  
}  
  
public void desenharArvore(boolean primeiraVez, NoArvore2<Aluno> atual,  
                           int x, int y, double angulo, double incremento,  
                           double comprimento, Graphics g)  
{  
    int xF, yF;  
    if (atual != null)  
    {  
        g.setColor(Color.red);  
        xF = (int) (x + Math.cos(angulo) * comprimento);  
        yF = (int) (y + Math.sin(angulo) * comprimento);  
  
        if (primeiraVez)  
            yF = 25;  
        g.drawLine(x,y,xF,yF);  
    }
```

9. Análise de Algoritmos

Por que seria desejável analisar um algoritmo? Estamos interessados em algoritmos que tenham sido precisamente especificados usando um formalismo matemático apropriado – tal como em uma linguagem de programação, ao invés de algoritmos escritos em linguagens naturais, como Português ou Inglês.

Dada uma tal expressão de um algoritmo, o que podemos fazer com ela? Obviamente podemos executar o programa e observar seu comportamento. Não é provável que isso seja muito útil ou informativo nos casos mais gerais. Se executarmos um programa específico em um computador com um conjunto específico de entradas, então tudo o que conheceremos é o comportamento do programa em uma única situação. Esse conhecimento é muito frágil e nós precisamos ser muito cuidadosos quando registramos conclusões baseadas em uma única evidência.

A fim de aprender mais sobre um algoritmo, podemos analisá-lo. Isto significa estudar a especificação do algoritmo e chegar a conclusões sobre como a sua implementação – o programa – irá comportar-se em geral.

O que podemos analisar em um algoritmo? Podemos determinar:

- o tempo de execução de um programa em função de suas entradas;
- o espaço máximo ou total de memória necessária para os dados do programa;
- o tamanho total do código do programa;
- se o programa calcula corretamente o resultado desejado;
- a complexidade do programa – isto é, quão fácil é sua leitura, entendimento e manutenção;
- a robustez do programa – isto é, como ele lida com entradas inesperadas ou errôneas.

Neste texto, estamos preocupados primariamente com o tempo de execução. Consideraremos também o espaço de memória necessário para executar o programa. Existem muitos fatores que afetam o tempo de execução de um programa. Entre eles estão o próprio algoritmo, os dados de entrada e o sistema computacional usado para executar o programa.

O desempenho de um computador é determinado por:

- o hardware:
 - processador usado (tipo e velocidade),
 - memória disponível (cache e RAM), e
 - disco disponível;
- a linguagem de programação na qual o algoritmo foi especificado;
- o compilador ou interpretador dessa linguagem que usamos;
- o software do sistema operacional do computador.

Uma análise detalhada do desempenho de um programa, que leve em conta todos esses fatores, é uma tarefa muito difícil e que consumiria muito tempo. Além disso, tal análise não terá uma significância duradoura. O rápido ritmo de mudanças nas tecnologias subjacentes significa que os resultados dessas análises não serão aplicáveis na próxima geração de hardware ou de software.

A fim de superar esta situação, criaremos um “modelo” do comportamento de um computador com os objetivos de simplificar a análise bem como produzir resultados significantes.

Um modelo detalhado do Computador

Nesta seção desenvolveremos um modelo detalhado do desempenho em tempo de execução de programas C#. O modelo desenvolvido será independente do hardware e do sistema operacional subjacentes. Ao invés de analisar o desempenho de uma máquina física específica, modelaremos a execução do programa C# na CLR (Common Language Runtime), que é um código intermediário entre a compilação e a execução e, portanto, independe de processadores ou sistemas específicos.

Uma consequência direta desta abordagem é que perderemos alguma confiabilidade – o modelo resultante poderá não prever acuradamente o desempenho do algoritmo em todos os possíveis sistemas de hardware/software. Por outro lado, o modelo resultante ainda é bastante complexo e rico em detalhes.

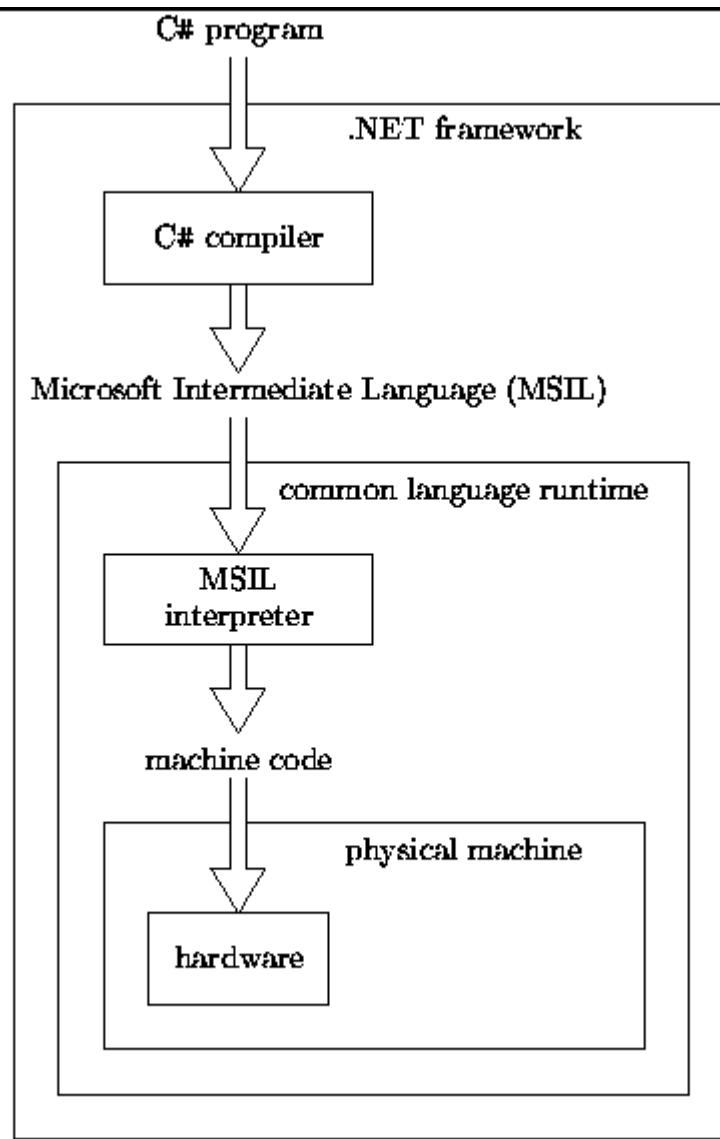


Figure 5.1: Sistema de execução do C#

Os axiomas básicos

O desempenho em tempo de execução da linguagem comum de tempo de execução (CLR) é dado por um conjunto de axiomas que deveremos postular agora. O primeiro axioma aborda o tempo de execução da referência a variáveis simples:

Axioma 1 - O tempo exigido para recuperar um operando da memória é uma constante, T_{fetch} , e o tempo exigido para armazenar um resultado na memória é uma constante, T_{store} .

De acordo com o Axioma 1, o comando de atribuição

$y = x;$

possui tempo de execução $T_{\text{fetch}} + T_{\text{store}}$. Isto é, o tempo gasto para recuperar o valor da variável x é T_{fetch} e o tempo gasto para armazenar esse valor na variável y é T_{store} .

Podemos aplicar o Axioma 1 para constantes também. Portanto, a atribuição

$y = 1;$

também possui tempo de execução $T_{\text{fetch}} + T_{\text{store}}$. Para confirmar por que essa afirmação está correta, considere que uma constante geralmente precisa ser armazenada na memória do computador, e podemos esperar que o “custo” de recuperá-la seja o mesmo que o de recuperar qualquer outro operando simples.

O próximo axioma endereça o tempo de execução de operações aritméticas simples:

Axioma 2 – Os tempos exigidos para realizar operações aritméticas elementares, tais como adição, subtração, multiplicação, divisão e comparações são todos constantes. Esses tempos são denotados por τ_+ , τ_- , τ_{\times} , τ_{\div} e τ_{\leq} , respectivamente.

De acordo com o Axioma 2, todas as operações simples podem ser realizadas numa quantidade fixa de tempo. Para que isso seja factível, o número de bits usados para representar um valor deve ser fixo. Em C#, o número de bits necessários para representar um valor varia de 8 (para byte e sbyte) até 64 (para long, ulong e double). Precisamente por esse número de bits usados ser sempre fixo é que podemos dizer que os tempos de execução são também fixos. Se números com tamanhos arbitrários são permitidos, então as operações aritméticas básicas também necessitariam de uma quantidade arbitrariamente longa de tempo para serem realizadas.

Pela aplicação dos Axiomas 1 and 2, podemos determinar que o tempo de execução de um comando como

$y = y + 1;$

é $2T_{\text{fetch}} + \tau_+ + T_{\text{store}}$. Isto ocorre porque precisamos recuperar dois operandos (y e 1), somá-los e, por fim, armazenar o resultado novamente em y .

A sintaxe de C# fornece várias formas alternativas de expresser o mesmo cálculo:

```
y += 1;  
++y;  
y++;
```

Assumiremos que essas alternativas exigem exatamente a mesma quantidade de tempo de execução exigida pelo comando original.

O terceiro axioma básico endereça o gasto adicional (overhead) para chamada e retorno de métodos:

Axioma 3 – O tempo exigido para chamar um método é uma constante, τ_{call} , e o tempo exigido para retorno de um método é uma constante, τ_{return} .

Quando um método é chamado, algumas operações de limpeza e arrumação precisam ser realizadas. Geralmente isto inclui salvar o endereço de retorno de tal forma que a execução do programa possa continuar no lugar correto após a chamada, salvar o estado de quaisquer cálculos parcialmente completos de forma que eles possam prosseguir após a chamada, e a alocação de um novo contexto de execução (quadro de pilha ou registro de ativação) no qual o método chamado possa ser avaliado. Simetricamente, quando do retorno de um método, todo esse trabalho é desfeito. Enquanto o overhead de chamada e retorno possa ser bem grande, de qualquer forma ele realiza uma quantidade constante de trabalho.

Além do overhead de chamada e de retorno, quando parâmetros são passados para o método temos um overhead adicional:

Axioma 4 – O tempo exigido para passar um argumento para um método é o mesmo tempo exigido para armazenar um valor na memória, τ_{store} .

A razão para tornar o overhead associado com a passagem de parâmetros ser o mesmo que o tempo necessário para armazenar um valor na memória é que a passagem de um argumento é, conceitualmente, o mesmo que atribuir o valor do parâmetro real para o parâmetro formal do método.

De acordo com o Axioma 4, o tempo de execução do comando

$$y = F(x);$$

será $\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{call}} + T_{F(x)}$, onde $T_{F(x)}$ é o tempo de execução do método F para a entrada x.

O primeiro dos dois armazenamentos é devido à passagem do parâmetro x para o método F; o segundo é proveniente da atribuição à variável y.

Um exemplo simples – Somatória de Séries Aritméticas

Nesta seção aplicaremos os axiomas 1, 2 e 3 para analisar o tempo de execução de um programa que calcula a série simples de somas abaixo :

$$\sum_{i=1}^n i.$$

O algoritmo que calcula essa somatória é dado no programa 5.1.

```

1 public class Example
2 {
3     public static int Sum(int n)
4     {
5         int result = 0;
6         for (int i = 1; i <= n; ++i)
7             result += i;
8         return result;
9     }
10 }
```

Programa 5.1: Programa para calcular $\sum_{i=1}^n i$.

Os comandos executáveis do programa 5.1 correspondem às linhas 5 a 8. A Tabela 5.1 especifica os tempos de execução de cada um desses comandos.

Comando	Tempo	Código
5	$T_{\text{fetch}} + T_{\text{store}}$	result = 0
6a	$T_{\text{fetch}} + T_{\text{store}}$	i = 1
6b	$(2T_{\text{fetch}} + \tau_<) \times (n + 1)$	i <= n
6c	$(2T_{\text{fetch}} + \tau_+ + T_{\text{store}}) \times n$	++i
7	$(2T_{\text{fetch}} + \tau_+ + T_{\text{store}}) \times n$	result += i
8	$T_{\text{fetch}} + T_{\text{return}}$	return result
TOTAL	$(6T_{\text{fetch}} + 2T_{\text{store}} + \tau_< + 2\tau_+) \times n$ + $(5T_{\text{fetch}} + 2T_{\text{store}} + \tau_< + T_{\text{return}})$	

Tabela 5.1: Calculando o tempo de execução do Programa 5.1.

Note que o comando `for` na linha 6 do Programa 5.1 foi separado em 3 linhas na tabela 5.1. Isto ocorreu para analisarmos o tempo de execução de cada um dos elementos de um comando `for` separadamente. O primeiro elemento, o *código de inicialização*, é executado uma vez antes da primeira iteração do loop. O segundo elemento, o *teste de término do loop*, é executado antes que cada iteração do loop seja iniciada. O número de vezes que o teste de término é executado é um a mais que o número de vezes que o corpo do loop é executado. Finalmente, o terceiro elemento, o *contador de passos do loop*, é executado uma vez por iteração do loop.

Somando as entradas na Tabela 5.1 obtemos que o tempo de execução, $T(n)$, do Programa 5.1 é

$$T(n) = t_1 + t_2 n \quad (2.1)$$

onde $t_1 = 5T_{\text{fetch}} + 2T_{\text{store}} + \tau_< + T_{\text{return}}$ e $t_2 = 6T_{\text{fetch}} + 2T_{\text{store}} + \tau_< + 2\tau_+$.

Operações de Indexação de Vetor

Agora abordaremos a questão do acesso dos elementos de um vetor de dados. Em geral, os elementos de um vetor unidimensional são armazenados em posições consecutivas da memória. Portanto, dado o endereço do primeiro elemento do vetor, uma simples adição é suficiente para determinar o endereço de um elemento qualquer do vetor:

Axioma 5 – O tempo exigido para o cálculo de endereço implicado numa operação de indexação de vetor, por exemplo, $a[i]$, é uma constante, $\tau[.]$. Esse tempo não inclui o tempo de cálculo da expressão de indexação, nem o tempo de acesso (recuperação ou armazenamento) do elemento do vetor.

Pela aplicação do Axioma 5, podemos determinar que o tempo de execução do comando

$$y = a[i];$$

é $3\tau_{\text{fetch}} + \tau[.] + \tau_{\text{store}}$. Três recuperações de operandos são necessárias: a primeira para recuperar a , que é o endereço base do vetor na memória, a segunda para recuperar i , que é o índice do vetor, e a terceira para recuperar $a[i]$, o elemento do vetor.

Outro Exemplo - Regra de Horner

Nesta seção aplicaremos os Axiomas 1, 2, 3 e 4 para analisar o tempo de execução de um programa que calcula o valor de um polinômio.

Isto é, dado os $n+1$ coeficientes a_0, a_1, \dots, a_n , e o valor x , desejamos calcular a seguinte somatória:

$$\sum_{i=0}^n a_i x^i.$$

A maneira usual de avaliar tais polinômios é usar a Regra de Horner, que é um algoritmo para calcular a somatória sem exigir o cálculo de potências arbitrárias de x . O algoritmo para calcular essa somatória é dada no Programa 5.2. Tabela 5.2 relaciona os tempos de execução de cada um dos comandos executáveis no Programa 5.2.

```

1 public class Example
2 {
3     public static int Horner(int[] a, int n, int x)
4     {
5         int result = a[n];
6         for (int i = n - 1; i >= 0; --i)
7             result = result * x + a[i];
8         return result;
9     }
10 }
```

Program 5.2: Programa para calcular $\sum_{i=0}^n a_i x^i$ usando a regra de Horner.

Comando	Tempo
5	$3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}$
6a	$2\tau_{\text{fetch}} + \tau_- + \tau_{\text{store}}$
6b	$(2\tau_{\text{fetch}} + \tau_+) \times (n + 1)$
6c	$(2\tau_{\text{fetch}} + \tau_- + \tau_{\text{store}}) \times n$
7	$(5\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_+ + \tau_x + \tau_{\text{store}}) \times n$
8	$\tau_{\text{fetch}} + \tau_{\text{return}}$
TOTAL	$(9\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_+ + \tau_{[\cdot]} + \tau_+ + \tau_x + \tau_-) \times n$ $+ (8\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{[\cdot]} + \tau_- + \tau_+ + \tau_{\text{return}})$

Tabela 5.2: Calculando o tempo de execução do Programa 5.2.

Somando as entradas na Tabela 5.2 obtemos que o tempo de execução, $T(n)$, do Programa 5.2 é

$$T(n) = t_1 + t_2 n \quad (2.2)$$

onde $t_1 = 8\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{[\cdot]} + \tau_- + \tau_+ + \tau_{\text{return}}$

e $t_2 = 9\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_+ + \tau_{[\cdot]} + \tau_+ + \tau_x + \tau_-$.

Analisando Métodos Recursivos

Nesta seção analisamos o desempenho de um algoritmo recursivo que calcula o fatorial de um número. Recorde que o fatorial de um inteiro n , não-negativo, denotado como $n!$, é definido por

$$n! = \begin{cases} 1 & n = 0, \\ \prod_{i=1}^n i & n > 0. \end{cases} \quad (2.3)$$

No entanto, também podemos definir o fatorial *recursivamente* como se segue

$$n! = \begin{cases} 1 & n = 0, \\ n \times (n-1)! & n > 0. \end{cases}$$

É esta última definição que leva ao algoritmo dado no Programa 5.3 para calcular o fatorial de n . A Tabela 5.3 fornece os tempos de execução de cada um dos comandos executáveis no Programa 5.3..

```

1 public class Example
2 {
3     public static int Factorial(int n)
4     {
5         if (n == 0)
6             return 1;
7         else
8             return n * Factorial(n - 1);
9     }
10 }
```

Programa 5.3 : Programa recursivo para calcular $n!$.

Comando	Tempo	
	$n=0$	$n>0$
5	$2\tau_{\text{fetch}} + \tau_{\text{return}}$	$2\tau_{\text{fetch}} + \tau_{\text{return}}$
6	$\tau_{\text{fetch}} + \tau_{\text{return}}$	--
8	--	$3\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{store}} + \tau_{x}$ $+ \tau_{\text{call}} + \tau_{\text{return}} + T(n - 1)$

Tabela 5.3: Cálculo do tempo de execução do Programa 5.3.

Note que analisamos separadamente o tempo de execução de dois possíveis resultados do teste condicional na linha 5. Claramente, o tempo de execução do programa depende do resultado deste teste.

Além disso, o método `Factorial` chama a si mesmo recursivamente na linha 8. Portanto, para escrevermos o tempo de execução da linha 8, precisamos conhecer o tempo de execução, $T(\cdot)$, de `Factorial`. Mas isto é precisamente o que estamos tentando determinar em primeiro lugar!

Saímos dessa cilada assumindo que já sabemos o que é a função $T(\cdot)$, e que podemos usar essa função para determinar o tempo de execução da linha 8.

Somando as colunas da tabela 5.3 obtemos que o tempo de execução do Programa 5.3 é

$$T(n) = \begin{cases} t_1 & n = 0, \\ T(n - 1) + t_2 & n > 0, \end{cases} \quad (2.4)$$

Onde $t_1 = 3\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{return}}$ e $t_2 = 5\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{return}} + \tau_{\text{store}} + \tau_{x} + \tau_{\text{call}} + \tau_{\text{return}}$.

Este tipo de equação é chamada uma *relação de recorrência* porque a função é definida em termos de si própria.

Resolvendo Relações de Recorrência – Substituição Repetida

Nesta seção apresentamos uma técnica para resolver uma relação de recorrência como a da Equação 2.4 chamada *substituição repetida*. A idéia básica é esta:

Dado que $T(n) = T(n - 1) + t_2$, queremos também escrever $T(n - 1) = T(n - 2) + t_2$, desde que $n > 1$.

Já que $T(n-1)$ aparece no lado direito da equação anterior, nós podemos substituir por ela todo o lado direito da última equação. Pela repetição desse processo obtemos

$$\begin{aligned} T(n) &= T(n - 1) + t_2 \\ &= (T(n - 2) + t_2) + t_2 \\ &= T(n - 2) + 2t_2 \\ &= (T(n - 3) + t_2) + 2t_2 \\ &= T(n - 3) + 3t_2 \\ &\vdots \end{aligned}$$

O passo seguinte exige um pouco de intuição : devemos tentar discernir o padrão que está emergindo. Neste caso ele é óbvio:

$$T(n) = T(n - k) + kt_2,$$

onde $1 \leq k \leq n$. Claramente, se nós temos dúvidas a respeito de nossa intuição, podemos sempre verificar nosso resultado através da indução matemática:

Prova (por Indução)

Caso Base : claramente a fórmula está correta para $k=1$, pois

$$T(n) = T(n - k) + kt_2 = T(n - 1) + t_2$$

Hipótese Indutiva

Suponha que $T(n) = T(n - k) + kt_2$ para $k = 1, 2, \dots, l$. Por esta suposição temos que

$$T(n) = T(n - l) + lt_2. \quad (2.5)$$

Note também que usando a relação de recorrência original podemos escrever

$$T(n - l) = T(n - l - 1) + t_2 \quad (2.6)$$

para $l \leq n$. Substituindo a Equação 2.6 no lado direito da Equação 2.5 resulta em

$$\begin{aligned} T(n) &= T(n-l-1) + t_2 + lt_2 \\ &= T(n-(l+1)) + (l+1)t_2 \end{aligned}$$

Portanto, por indução em l , nossa fórmula é correta para todo valor k tal que $0 \leq k \leq n$.

Portanto, demonstramos $T(n) = T(n-k) + kt_2$ que, para $1 \leq k \leq n$. Agora, se n era conhecido, poderíamos repetir o processo de substituição até que obtivéssemos $T(0)$ no lado direito. O fato de n ser desconhecido não deveria nos deter – pois nós obtemos $T(0)$ no lado direito quando $n-k=0$. Isto é, quando $k=n$. Fazendo $k=n$ nós obtemos

$$\begin{aligned} T(n) &= T(n-k) + kt_2 \\ &= T(0) + nt_2 \\ &= t_1 + nt_2 \end{aligned} \tag{2.7}$$

onde $t_1 = 3\tau_{\text{fetch}} + \tau_{\leftarrow} + \tau_{\text{return}}$ e $t_2 = 5\tau_{\text{fetch}} + \tau_{\leftarrow} + \tau_{\rightarrow} + \tau_{\text{store}} + \tau_x + \tau_{\text{call}} + \tau_{\text{return}}$.

Outro Exemplo-Encontrar o maior elemento de um vetor

Nesta seção consideraremos o problema de encontrar o maior elemento de um vetor. Isto é, dado um vetor de n inteiros não-negativos, a_0, a_1, \dots, a_{n-1} , nós desejamos encontrar

$$\max_{0 \leq i < n} a_i.$$

A maneira mais direta de resolver esse problema é realizar uma *busca linear* do vetor. O algoritmo de busca linear é dado no Programa 5.4 e os tempos de execução para os vários comandos são dados na Tabela 5.4.

```

1 public class Example
2 {
3     public static int FindMaximum(int[] a)
4     {
5         int result = a[0];
6         for (int i = 1; i < a.Length; ++i)
7             if (a[i] > result)
8                 result = a[i];
9         return result;
10    }
11 }
```

Program 5.4 : Busca linear para encontrar $\max_{0 \leq i < n} a_i$

Comando	Tempo
5	$3\tau_{\text{fetch}} + \tau_{\leftarrow} + \tau_{\text{store}}$
6a	$\tau_{\text{fetch}} + \tau_{\text{store}}$
6b	$(2\tau_{\text{fetch}} + \tau_{\leftarrow}) \times n$

6c	$(2\tau_{\text{fetch}} + \tau_+ + \tau_{\text{store}}) \times (n - 1)$
7	$(4\tau_{\text{fetch}} + \tau_{[.]} + \tau_-) \times (n - 1)$
8	$(3\tau_{\text{fetch}} + \tau_{[.]} + \tau_{\text{store}}) \times ?$
9	$\tau_{\text{fetch}} + \tau_{\text{store}}$

Tabela 5.4: Calculando o tempo de execução do Program 5.4.

Com a exceção da linha 8, os tempos de execução seguem simplesmente dos Axiomas 1, 2 e 5. Especificamente, observe que o corpo do loop é executado $n-1$ vezes. Isto significa que o teste de condição na linha 7 é executado $n-1$ vezes. No entanto, o número de vezes em que a linha 8 é executada depende da disposição e do conteúdo dos dados armazenados no vetor e não somente de n .

Se considerarmos que em cada iteração do corpo do loop, a variável result conterá o maior elemento do vetor encontrado até o presente momento, então a linha 8 será executada na i -ésima iteração do loop somente se a_i satisfaz o seguinte

$$a_i > \left(\max_{0 \leq j < i} a_j \right).$$

Portanto, o tempo de execução do Program 5.4, $T(\cdot)$, é uma função não somente do número de elementos no vetor, n , mas também dos valores armazenados no vetor, a_0, a_1, \dots, a_{n-1} . Somando as entradas da Tabela 5.4 obtemos

$$T(n, a_0, a_1, \dots, a_{n-1}) = t_1 + t_2 n + \sum_{i=1}^{n-1} t_3$$

$a_i > \left(\max_{0 \leq j < i} a_j \right)$

onde

$$\begin{aligned} t_1 &= 2\tau_{\text{store}} - \tau_{\text{fetch}} - \tau_+ - \tau_- \\ t_2 &= 8\tau_{\text{fetch}} + 2\tau_- + \tau_{[.]} + \tau_+ + \tau_{\text{store}} \\ t_3 &= 3\tau_{\text{fetch}} + \tau_{[.]} + \tau_{\text{store}}. \end{aligned}$$

Enquanto este resultado pode ser correto, ele não é nada útil. A fim de determinar o tempo de execução desse programa nós precisamos conhecer o número de elementos no vetor, n , e também conhecer os valores dos elementos do vetor, a_0, a_1, \dots, a_{n-1} . Até mesmo que conheçamos esses dados, verifica-se que, para calcular o tempo de execução do algoritmo, $T(n, a_0, a_1, \dots, a_{n-1})$, nós temos realmente que resolver o problema original, pois ele depende de n , dos valores armazenados e da disposição desses valores dentro do vetor.

Em outras palavras, para cada vez que executamos esse programa, poderemos ter quantidades diferentes (mudança em n), ou valores diferentes ou disposições diferentes dos mesmos valores. Sendo assim, cada execução será um novo problema a ser resolvido e não se pode, a priori, calcular-se o tempo gasto na execução sem conhecer-se esses 3 fatores e, pior ainda, sem executar o algoritmo totalmente.

Para podermos chegar a alguma conclusão útil sobre a eficiência desse algoritmo, precisaríamos executá-lo múltiplas vezes, variando n , os valores e sua disposição, tomando os tempos gastos e calculando um valor médio.

Após várias execuções com valores aleatórios para n , para os dados e para a disposição dos dados dentro do vetor, poderíamos chegar à conclusão, na média de todas as execuções (em uma quantidade estatisticamente significativa para análise), de que a linha 8 seria executada em metade dos acessos aos elementos do vetor e que teríamos de, em média, percorrer a metade dos n elementos do vetor para encontrar o maior valor.

No entanto, se o vetor estivesse com seus valores ordenados crescentemente, e não aleatoriamente distribuídos, o maior valor estaria na última posição e, portanto, bastaria um único acesso para encontrá-lo. Com esse exemplo percebe-se claramente que a velocidade de execução é uma função da disposição dos dados, nesse algoritmo.

Tempos de Execução Médios

Na seção anterior, encontramos a função $T(n, a_0, a_1, \dots, a_{n-1})$, que resulta no tempo de execução do programa 5.4 como uma função tanto do número de entradas, n , e dos valores específicos dos valores de entrada. Suponha que, ao contrário, estamos interessados em uma função $T_{\text{average}}(n)$ que nos dá o tempo de execução *médio* para n entradas, sem levar em conta os valores de entrada. Em outras palavras, se executarmos o programa 5.4, por um grande número de vezes numa seleção de valores aleatórios de tamanho n , qual será o tempo médio de execução?

POdemos escrever a soma dos tempos de execução dados na Tabela 5.4 da seguinte forma

$$T_{\text{average}}(n) = t_1 + t_2n + \sum_{i=1}^{n-1} p_i t_3 \quad (2.8)$$

onde p_i é a probabilidade de que a linha 8 do programa seja executada. A probabilidade p_i é dada por

$$p_i = P \left[a_i > (\max_{0 \leq j < i} a_j) \right].$$

Ou seja, p_i é a probabilidade que a i -ésima posição do vetor, a_i , seja maior que o maior valor de todas as posições precedentes do vetor, a_0, a_1, \dots, a_{i-1} .

Para determinar p_i , precisamos conhecer (ou supor) algo sobre a distribuição dos valores de entrada. Por exemplo, se sabemos *a priori* que o vetor passado para o método `FindMaximum` é ordenado do menor par ao maior, então sabemos que $p_i = 1$ (100% de probabilidade, para todo $1 \leq i \leq n$). De forma oposta, se sabemos que o vetor está ordenado do maior para o menor (decreasingemente), saberemos também que $p_i = 0$.

No caso mais geral, não temos um conhecimento, *a priori*, da distribuição dos valores no vetor. Neste caso, considere a i -ésima iteração do loop. Nesta iteração a_i é comparado com o maior de todos os i valores precedentes no vetor, a_0, a_1, \dots, a_{i-1} . A linha 6 of Programa 5.4 somente sera executada se a_i é o maior dos $i+1$ valores a_0, a_1, \dots, a_i . Não havendo nenhuma restrição imposta, podemos dizer que isto acontecerá com probabilidade $1/(i+1)$. Portanto

$$\begin{aligned} p_i &= P \left[a_i > (\max_{0 \leq j < i} a_j) \right] \\ &= \frac{1}{i+1}. \end{aligned} \tag{2.9}$$

Substituindo esta expressão para p_i na Equation 2.8 e simplificando o resultado, nós obtemos

$$\begin{aligned} T_{\text{average}}(n) &= t_1 + t_2 n + \sum_{i=1}^{n-1} p_i t_3 \\ &= t_1 + t_2 n + t_3 \sum_{i=1}^{n-1} \frac{1}{i+1} \\ &= t_1 + t_2 n + t_3 \left(\sum_{i=1}^n \frac{1}{i} - 1 \right) \\ &= t_1 + t_2 n + t_3 (H_n - 1) \end{aligned} \tag{2.10}$$

onde $H_n = \sum_{i=1}^n \frac{1}{i}$, é o n -ésimo *número harmônico*.

Sobre Números Harmônicos

A série $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots$ é chamada de *série harmônica*, e a somatória

$$H_n = \sum_{i=1}^n \frac{1}{i}$$

gera a série de *números harmônicos*, H_1, H_2, \dots . Como se poderá perceber, números harmônicos aparecem frequentemente na análise de algoritmos. Portanto, seria importante conhecer um pouco sobre como eles se comportam.

Uma característica notável dos números harmônicos é que, mesmo enquanto n se torna maior e a diferença entre números harmônicos consecutivos vai-se tornando arbitrariamente pequena

($H_n - H_{n-1} = \frac{1}{n}$), a série não converge! Isto é, $\lim_{n \rightarrow \infty} H_n$ não existe.

Em outras palavras, a somatória $\sum_{i=1}^{\infty} \frac{1}{i}$ vai ao infinito, mas apenas levemente.

A Figura 5.2 nos auxilia a entender o comportamento dos números harmônicos. A curva suave neta figura é a função $y=1/x$. Os degraus descendentes representam a função $y=1/\lfloor x \rfloor$ (divisão apenas por números inteiros).

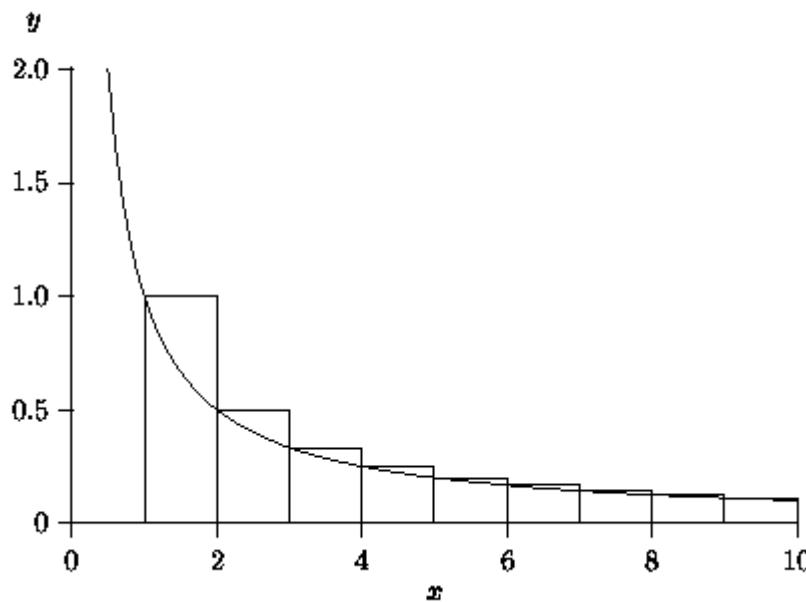


Figura 5.2: Calculando números harmônicos.

Note que a área sob os degraus entre 1 e n para qualquer inteiro $n > 1$ é dada por

$$\begin{aligned} \int_1^n \frac{1}{[x]} dx &= \sum_{i=1}^{n-1} \frac{1}{i} \\ &= H_{n-1}. \end{aligned}$$

Assim, se pudermos determinar a área sob os degraus descendentes na Figura 5.2, poderemos determinar os valores dos números harmônicos.

Como uma aproximação, considere a área sob a curva suave $y = 1/x$:

$$\begin{aligned} \int_1^n \frac{1}{x} dx &= \ln x \Big|_1^n \\ &= \ln(n). \end{aligned}$$

Portanto, H_{n-1} é aproximadamente $\ln n$ para $n > 1$.

Se aproximarmos H_{n-1} por $\ln n$, o erro nesta aproximação é igual à área entre as duas curvas (degrau e curva suave). De fato, a área entre essas duas curvas é uma quantidade de tal importância que tem seu próprio, γ , chamado de *constante de Euler*. A derivação a seguir indica uma maneira de calcular a constante de Euler:

$$\begin{aligned}
 \gamma &= \lim_{n \rightarrow \infty} (H_{n-1} - \ln n) \\
 &= \sum_{i=1}^{\infty} \left(\int_i^{i+1} \left(\frac{1}{x} - \frac{1}{i} \right) dx \right) \\
 &= \sum_{i=1}^{\infty} \left(\frac{1}{i} \int_i^{i+1} 1 dx - \int_i^{i+1} \frac{1}{x} dx \right) \\
 &= \sum_{i=1}^{\infty} \left(\frac{1}{i} - \ln \left(\frac{i+1}{i} \right) \right) \\
 &\approx 0.577215
 \end{aligned}$$

Um programa para calcular a constante de Euler na base dessa derivação é dada no Program 5.5. Enquanto essa não é necessariamente a maneira mais precisa ou mais veloz de calcular a constante de Euler, ela fornece um resultado correto até a sexta casa decimal.

```

1 public class Example
2 {
3     public static double Gamma()
4     {
5         double result = 0;
6         for (int i = 1; i <= 500000; ++i)
7             result += 1.0/i - Math.Log((i + 1.0)/i);
8         return result;
9     }
10 }
```

Programa 5.5: Programa para calcular γ .

Portanto, com a constante de Euler's em mãos, podemos escrever uma expressão para o $(n-1)^{\text{th}}$ número harmônico:

$$H_{n-1} = \ln n + \gamma - \epsilon_n \quad (2.11)$$

onde ϵ_n é o erro introduzido pelo fato de que γ é definido como a diferença entre as curvas do intervalo $[1, +\infty)$, mas nós somente precisamos da diferença no intervalo $[1, n]$. Como resultado, pode-se demonstrar (mas não aqui), que existe uma constante K tal que para valores muito grandes de n ,²

² Guillaume François Antoine de L'Hôpital, Marquês de Sainte-Mesme, é conhecido por sua regra para cálculo de limites que estabelece que

Se $\lim_{x \rightarrow p} f(x) = 0$ e $\lim_{x \rightarrow p} g(x) = 0$, então, se existir $\lim_{x \rightarrow p} \frac{f'(x)}{g'(x)}$, então:

Desde que o termo de erro é menor que $1/n$, podemos adicionar $1/n$ em ambos os lados da Equação 2.11 e ainda teremos um erro que vai se aproximando de zero conforme n se torna maior. Portanto, a aproximação usual para o número harmônico é

$$H_n \approx \ln n + \gamma.$$

Agora retornaremos à questão de encontrar o tempo de execução médio do Programa 5.4, que encontra o maior elemento de um vetor. Podemos agora reescrever a Equação 2.10 para resultar em

$$\begin{aligned} T_{\text{average}}(n) &= t_1 + t_2 n + t_3(H_n - 1) \\ &\approx t_1 + t_2 n + t_3(\ln n + \gamma - 1) \\ &\approx (t_1 + t_3(\gamma - 1)) + t_2 n + t_3 \ln n. \end{aligned}$$

O último Axioma

Nesta seção estabelecemos o último axioma necessário para o modelo detalhado da CLR. Este axioma aborda o tempo exigido para criar uma nova instância de um objeto:

Axioma 6 – O tempo exigido para criar uma nova instância de um objeto usando o operador `new` é uma constante, τ_{new} . Este tempo não inclui qualquer tempo gasto para inicializar o objeto.

Pela aplicação dos Axiomas 1, 3, 4 e 6, podemos determinar que o tempo de execução do comando

```
Int32 ref = new Int32(0);
```

É $\tau_{\text{new}} + \tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{call}} + \mathcal{T}\langle \text{Int32}() \rangle$, onde $\mathcal{T}\langle \text{Int32}() \rangle$ é o tempo de execução do construtor de `Int32`.

$$\lim_{x \rightarrow p} \frac{f(x)}{g(x)} = \lim_{x \rightarrow p} \frac{f'(x)}{g'(x)}$$