Apostila de Estrutura de Dados e Algoritmos em C#

Prof. Ms. Eduardo R. Marcelino

eduardormbr@gmail.com

2007/2020

Conteúdo

Referências básicas para o material desta apostila:

[1] PEREIRA, Silvio do Lago. Estruturas de dados fundamentais – Conceitos e aplicações. São Paulo: Érica, 1996.

[2] ZIVIANI, Nivio. Projeto de Algoritmos – com implementações em Pascal e C, 2. ed. São Paulo: Pioneira Thomson Learning, 2005.

[3] GOODRICH, M.T., **TAMASSIA**, R. Estruturas de Dados e Algoritmos em Java, 2. ed. Porto Alegre: Bookman, 2002.

[4] FORBELLONE, A.L.V., **EBERSPACHER**, H.F.**,** Lógica de Programação – A Construção de Alboritmos e Estrutura de Dados. 2. ed. São Paulo: 2000. Makron Books.

[5] MORAES, C.R. Estruturas de Dados e Algoritmos – Uma abordagem didática. São Paulo: 2001. Berkeley.

TIPOS ABSTRATOS DE DADOS (TAD)

É formado por um conjunto de valores e por uma série de funções que podem ser aplicadas sobre estes valores. Funções e valores, em conjunto, constituem um modelo matemático que pode ser empregado para "modelar" e solucionar problemas do mundo real, servido para especificar as características relevantes dos objetos envolvidos no problema, de que forma eles se relacionam e como podem ser manipulados. O TAD define **o que** cada operação faz, mas não **como** o faz.

EX: **TAD** para uma **PILHA**:

Empilha (valor) – Insere um valor no topo da pilha
Entrada: valor. *Saída*: nenhuma.

Desempilha - Retira um valor do topo da pilha e o devolve.
Entrada: nenhuma. *Saída*: valor.

TIPOS DE DADOS CONCRETOS (TDC)

Sendo o TAD apenas um modelo matemático, sua definição não leva em consideração como os valores serão representados na memória do computador, nem se preocupa com o "tempo" que será gasto para aplicar as funções (rotinas) sobre tais valores. Sendo assim, é preciso transformar este TAD em um **tipo de dados concreto**. Ou seja, precisamos implementar (programar) as funções definidas no TAD. É durante o processo de implementação que a estrutura de armazenamento dos valores é especificada, e que os algoritmos que desempenharão o papel das funções são projetados.

Tipo Abstrato de Dados ⇒ Implementação ⇒ Tipo de Dados Concreto

LIMITAÇÕES DE UMA IMPLEMENTAÇÃO

É importante notar que nem todo TAD pode ser implementado em toda sua generalidade. Imagine por exemplo um TAD que tenha como função mapear todos os números primos. Claramente, este é um tipo de dados abstrato que não pode ser implementado universalmente, pois qualquer que seja a estrutura escolhida para armazenar os números primos, nunca conseguiremos mapear num espaço limitado de memória um conjunto infinito de valores.

Frequentemente, nenhuma implementação é capaz de representar um modelo matemático completamente; assim, precisamos reconhecer as limitações de uma implementação particular. Devemos ter em mente que podemos chegar a diversas implementações para um mesmo tipo de dados abstrato, cada uma delas apresentando vantagens e desvantagens em relação às outras. O projetista deve ser capaz de escolher aquela mais adequada para resolver o problema específico proposto, tomando como **medidas de eficiência** da implementação, sobretudo, as suas **necessidades de espaço de armazenamento** e **tempo de execução**. Abaixo, veja o quadro com a complexidade de alguns jogos:

Damas - 5 x 10 na potência 20 Cerca de 500.000.000.000.000.000.000 posições possíveis. Só em 1994 um programa foi capaz de vencer um campeão mundial. Em 2007 o jogo foi "solucionado" a ponto de ser possível um programa imbatível	**Poker americano (Texas hold'em)** - 10 na potência 18 Cerca de 1.000.000.000.000.000.000 posições possíveis. O campeonato mundial de humanos contra máquinas começou nesta semana, com favoritismo para os humanos. Um programa imbatível pode surgir nos próximos anos	**Xadrez** - 1 x 10 na potência 45 Cerca de 1.000.000.000.000.000.000.000.000.000.000.000.000.000.000 posições possíveis. Em 1997, um supercomputador venceu um campeão mundial depois de penar muito. Para criar um programa imbatível, porém, seria preciso de um computador quântico, uma máquina que por enquanto só existe na cabeça dos cientistas.

Trecho da entrevista com JONATHAN SCHAEFFER, **pesquisador na área de inteligência artificial.**

Disponível em: (acesso em 25/072007)
http://circuitointegrado.folha.blog.uol.com.br/arch2007-07-22_2007-07-28.html#2007_07-25_07_57_02-11453562-0

O jogo que utiliza o algoritmo criado por Jonathan está disponível em:

http://www.cs.ualberta.ca/~chinook/

FOLHA - Você pretende continuar trabalhando no problema para chegar ao que os matemáticos chamam de uma "solução forte", mapeando cada uma das posições do jogo?
SCHAEFFER - Não, por uma boa razão. Pondo em perspectiva, vemos que as damas possuem 5 x 10 na potência 20 posições. Muitas pessoas têm computadores com discos rígidos de 100 gigabytes [10 na potência 11 bytes]. Se você tiver uma máquina "hiper-super" você deve ter um disco de 1 terabyte [10 na potência 12]. Se você for a um dos 50 supercomputadores mais poderosos do mundo, você encontrará um disco de 1 petabyte [10 na potência 15]. Um disco rígido desses custa cerca de US$ 1 milhão. As damas têm 10 na potência 20 posições. Para poder gravar a solução forte do problema eu precisaria de 500 mil petabytes _o que custaria US$ 500 bilhões hoje. Acho que não é muito factível. Se eu processasse a solução, eu simplesmente não teria onde salvá-la.

FOLHA - Você acha possível que alguém encontre uma solução para xadrez como a que o sr. tem para damas? Quando isso vai ocorrer?
SCHAEFFER - Não consigo conceber alguém conseguindo isso por um longo tempo, a não ser que haja avanços fundamentais de grande impacto. O número de posições do xadrez é da ordem de 10 na potência 45. É até difícil de imaginar quão grande isso é. Não acho que isso possa ser feito com os computadores que temos hoje ou com os que teremos nas próximas décadas.
Há uma tecnologia experimental despontando, chamada computação quântica, que pode um dia vir a se tornar realidade. Talvez a computação quântica possa dar conta desse processamento, mas ainda há muito futuro até lá. Eu gostaria de viver tempo o suficiente para ver isso, mas não acho que vai ser possível.

Referências:
PEREIRA, Silvio do Lago. Estruturas de dados fundamentais – Conceitos e aplicações. São Paulo: Érica, 1996.

COMPLEXIDADE COMPUTACIONAL

Fontes:http://www.dca.fee.unicamp.br/~ting/Courses/ea869/faq1.html
http://pt.wikipedia.org/wiki/Complexidade_computacional
http://pt.wikipedia.org/wiki/Complexidade_quadr%C3%A1tica

O que é um problema computável?

Um problema é computável se existe um procedimento que o resolve em um número finito de passos, ou seja se existe um algoritmo que leve à sua solução.

Observe que um problema considerado "em princípio" computável pode não ser tratável na prática, devido às limitações dos recursos computacionais para executar o algoritmo implementado.

Por que é importante a análise de complexidade computacional de um algoritmo?

A complexidade computacional de um algoritmo diz respeito aos recursos computacionais - espaço de memória e tempo de máquina - requeridos para solucionar um problema.

Geralmente existe mais de um algoritmo para resolver um problema. A análise de complexidade computacional é portanto fundamental no processo de definição de algoritmos mais eficientes para a sua solução. Apesar de parecer contraditório, com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do "tamanho" dos problemas a serem resolvidos.

O que entendemos por tamanho de um problema?

O tamanho de um problema é o tamanho da entrada do algoritmo que resolve o problema. Vejamos os seguintes exemplos:

- A busca em uma lista de N elementos ou a ordenação de uma lista de N elementos requerem mais operações à medida que N cresce;
- O cálculo do fatorial de N tem o seu número de operações aumentado com o aumento de N;
- A determinação do valor de F_N na sequência de Fibonacci F_0, F_1, F_2, F_3, ... envolve uma quantidade de adições proporcional ao valor de N.

A Teoria da complexidade computacional é a parte da teoria da computação que estuda os recursos necessários durante o cálculo para resolver um problema. O termo foi criado pelo Juris Hartmanis e Richard Stearns[1]. Os recursos comumente estudados são o tempo (número de passos de execução de um algoritmo para resolver um problema) e o espaço (quantidade de memória utilizada para resolver um problema). Pode-se estudar igualmente outros parâmetros, tais como o número de processadores necessários para resolver o problema em paralelo. A teoria da complexidade difere da teoria da computabilidade a qual se ocupa de factibilidade de se solucionar um problema como algoritmos efetivos sem levar em conta os recursos necessários para ele.

Os problemas que têm uma solução com ordem de complexidade linear são os problemas que se resolvem em um tempo que se relaciona linearmente com seu tamanho.

Os problemas com custo fatorial ou combinatório estão agrupados em NP. Estes problemas não têm uma solução prática, significa dizer quer, uma máquina não pode resolver-los em um tempo razoável.

Análise assintótica

A importância da complexidade pode ser observada no exemplo abaixo, que mostra 5 algoritmos A1 a A5 para resolver um mesmo problema, de complexidades diferentes. Supomos que uma operação leva 1 milisegundo para ser efetuada. A tabela seguinte dá o tempo necessário por cada um dos algoritmos.

Tk(n) é a complexidade do algoritmo.

	A_1	A_2	A_3	A_4	A_5
n	$T_1(n) = n$	$T_2(n) = n\log n$	$T_3(n) = n^2$	$T_4(n) = n^3$	$T_5(n) = 2^n$
16	0,016s	0,064s	0,256s	4s	1m4s
32	0,032s	0,16s	1s	33s	46 dias
512	0,512s	9s	4m22s	1 dia 13h	10^{137} séculos

A **complexidade do tempo** de um problema é o número de passos que se toma para resolver uma instância de um problema, a partir do tamanho da entrada utilizando o algoritmo mais eficiente à disposição. Intuitivamente, caso se tome uma instância com entrada de longitude *n* que pode resolver-se em n^2 passos, se diz que esse problema tem uma complexidade em tempo de n^2. Supostamente, o número exato de passos depende da máquina em que se programa, da linguagem utilizada e de outros fatores. Para não ter que falar do custo exato de um cálculo se utiliza a notação assintótica. Quando um problema tem custo dado em tempo $O(n^2)$ em uma configuração de computador e linguagem, este custo será o mesmo em todos os computadores, de maneira que esta notação generaliza a noção de custo independentemente do equipamento utilizado.

Notação O, Ω (Omega) e Theta

São usadas três perspectivas no estudo do caso da complexidade:

Melhor Caso - Ω( )

Representado por Ω( ) (Ômega), consiste em assumir que vai acontecer o melhor. Pouco utilizado e de baixa aplicabilidade.
Exemplo 1: Em uma lista telefônica queremos encontrar um número, assume-se que a complexidade do caso melhor é Ω (1), pois está pressupondo-se o número desejado está no topo da lista.

Exemplo 2: Extrair qualquer elemento de um vetor. A indexação em um vetor ou array, leva o mesmo tempo seja qual for o índice que se queira buscar. Portanto é uma operação de complexidade constante Ω (1).

Pior Caso - O( )

Representado normalmente por O( ) (BIG O) . Se dissermos que um determinado algoritmo é representado por g(x) e a sua complexidade Caso Pior é n, será representada por g(x) = O(n). Consiste em assumir que o pior dos casos pode acontecer, sendo de grande utilização e também normalmente o mais fácil de ser determinado.

Exemplo 1: Será tomado como exemplo o jogo de azar com 3 copos, deve descobrir-se qual deles possui uma moeda debaixo dele, a complexidade caso pior será O(3) pois no pior dos casos a moeda será encontrada debaixo do terceiro copo, ou seja, será encontrada apenas na terceira tentativa.

Exemplo 2: O processo mais comum para ordenar um conjunto de elementos têm complexidade quadrática. O procedimento consiste em criar uma coleção vazia de elementos. A ela se acrescenta, em ordem, o menor elemento do conjunto original que ainda não tenha sido eleito, o que implica percorrer completamente o conjunto original (O(n), sendo *n* o número de elementos do conjunto). Esta percorrida sobre o conjunto original se realiza até que se tenha todos seus elementos na seqüência de resultado. Pode-se ver que há de se fazer *n* seleções (se ordena todo o conjunto) cada uma com um custo *n* de execução: o procedimento é de ordem quadrática $O(n^2)$. Deve esclarecer se de que há diversos algoritmos de ordenação com melhores resultados.

Caso médio - θ( )

Representado por θ( ) (Theta). Este é o caso que é o mais difícil de ser determinado, pois, necessita de análise estatística e em conseqüência de muitos testes, contudo é muito utilizado, pois é o que representa mais corretamente a complexidade do algoritmo.

Exemplo: Procurar uma palavra em um dicionário. Pode-se iniciar a busca de uma palavra na metade do dicionário. Imediatamente se sabe se foi encontrada a palavra ou, no caso contrário, em qual das duas metades deve se repetir o processo (é um processo recursivo) até se chegar ao resultado. Em cada busca (ou sub-busca), o problema (as páginas em que a palavra pode estar) vão se reduzindo à metade, o que corresponde com a função logarítmica. Este

procedimento de busca (conhecido como busca binária) em uma estrutura ordenada têm complexidade logarítmica $\theta(\log_2 n)$.

Exemplos de Complexidade

Complexidade Constante

Representada por O(1). Complexidade algorítmica cujo tempo de execução independe do número de elementos na entrada.

```
            Console.Write("Digite um texto: ");
            texto = Console.ReadLine();

            Console.Write("Digite uma letra: ");
            letra = Console.ReadLine()[0];
```

Apesar de termos 4 instruções, o processo como um todo será considerado como O(1), pois neste caso executar 1 ou 4 instruções é irrelevante.

Complexidade Linear - O(n)

Representada por O(*n*). Complexidade algorítmica em que um pequeno trabalho é realizado sobre cada elemento da entrada. Esta é a melhor situação possível para um algoritmo que tem que processar *n* elementos de entrada ou produzir *n* elementos de saída. Cada vez que *n* dobra de tamanho o tempo de execução dobra.

Programa exemplo: Determinar o maior elemento em um vetor:

```
        static void Main(string[] args)
        {
            int[] numeros = { 7, 5, 21, 33, 79, 8, 6, 29, 44, 10, -5 };

            int maior = numeros[0];

            for (int n = 1; n < numeros.Length; n++)
            {
                if (numeros[n] > maior)
                    maior = numeros[n];
            }

            Console.WriteLine(maior);
        }
```

A complexidade é dada por **n** elementos (o elemento que determina a taxa de crescimento do algoritmo acima) ou **O(n) (linear). Ou seja, depende da quantidade de elementos no vetor!**

Complexidade Logaritmica

Representada por O(log *n*). Complexidade algorítmica no qual algoritmo resolve um problema transformando-o em partes menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande. Por exemplo, quando *n* é um milhão, $log_2 n$ é aproximadamente 20.

Complexidade Quadrática - $O(n^2)$

Representada por O(n^2). Complexidade algorítmica que ocorrem quando os itens de dados são processados aos pares, muitas vezes em uma repetição dentro da outra. Por exemplo, quando n é mil, o número de operações é da ordem de 1 milhão. Algoritmos deste tipo somente são úteis para resolver problemas de tamanhos relativamente pequenos.

Programa exemplo: Preencher uma matriz NxN com números aleatórios.

```
static int[,] CriaMatriz(int tamanho)
{
    int[,] matriz = new int[tamanho, tamanho];

    Random r = new Random();

    //preenche a matriz NxN (onde N = tamanho) com números randomicos
    for (int i = 0; i < tamanho; i++)
        for (int j = 0; j < tamanho; j++)
            matriz[i, j] = r.Next(1000);

    return matriz;
}
```

A complexidade é dada por **n^2 (i*j)** elementos ou **$O(n^2)$** (quadrática)

Funções limitantes superiores mais conhecidas:

Melhor → Pior

Constante	Logarítmica	Linear	Quadrática	Exponencial
O(1)	O(log n)	O(n)	$O(n^2)$	$O(a^n)$ com a > 1

Exemplo de crescimento de várias funções

n	log n	n	n^2	2^n
2	1	2	4	4
4	2	4	16	16
8	3	8	64	256
16	4	16	256	65.536
32	5	32	1.024	4.294.967.296
64	6	64	4.096	$1,84 \times 10^{19}$
128	7	128	16.384	$3,40 \times 10^{38}$
256	8	256	65.536	$1,18 \times 10^{77}$
512	9	512	262.144	$1,34 \times 10^{154}$
1024	10	1024	1.048.576	$1,79 \times 10^{308}$

Tempos de execução dos algoritmos:

Os Algoritmos têm tempo de execução proporcional a:

O(**1)** - muitas instruções são executadas uma só vez ou poucas vezes (se isto for verdade para todo o programa diz-se que o seu tempo de execução é constante)

O(**log N)** - tempo de execução é logarítmico (cresce linearmente à medida que **N** cresce) (quando **N** duplica **log N** aumenta mas muito pouco;**)**

O(**N)** - tempo de execução é linear (típico quando algum processamento é feito para cada dado de entrada) (situação ótimas quando é necessário processar **N** dados de entrada) (ou produzir **N** dados na saída)

O (**N log N)** - típico quando se reduz um problema em subproblemas, se resolve estes separadamente e se combinam as soluções (se **N** é 1 milhão, **N log N** é perto de 20 milhões)

O (N^2) - tempo de execução típico **N** quadrático (quando é preciso processar todos os pares de dados de entrada) (prático apenas em pequenos problemas). EX: para **N**=10, $\mathbf{N^2}$ = 100.

O (N^3) - tempo de execução cúbico. EX: para **N** = 100, $\mathbf{N^3}$ = 1 milhão

O (2^N) - tempo de execução exponencial (provavelmente de pouca aplicação prática; típico em soluções de força bruta) (para **N** = 20, $\mathbf{2^N}$ = 1 milhão)

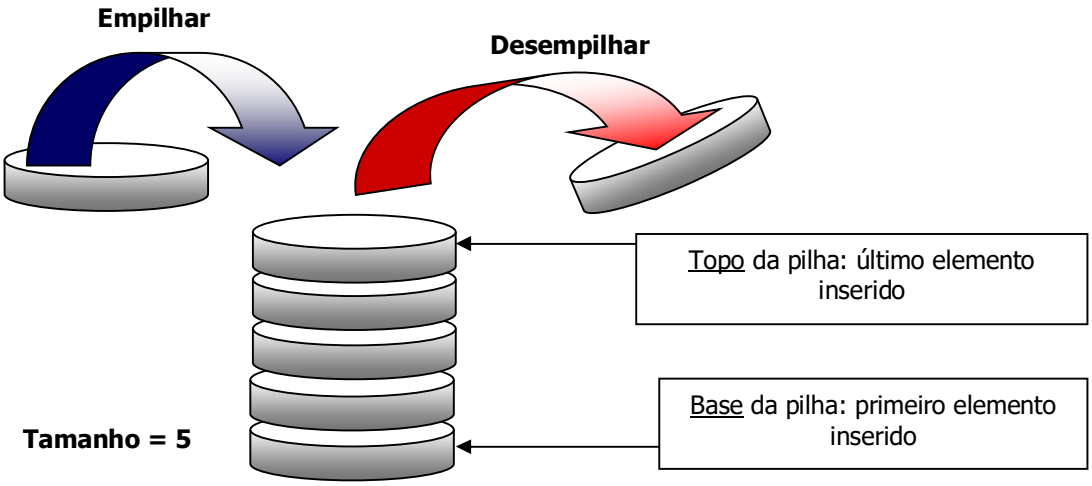
PILHAS

(http://pt.wikibooks.org/wiki/Estrutura_de_Dados_II/Pilhas)

Uma pilha é uma estrutura de dados onde em todas as inserções, retiradas e acessos ocorrem apenas em um dos extremos (no caso, em seu topo).

Os elementos são removidos na ordem inversa daquela em que foram inseridos de modo **que o último elemento que entra é sempre o primeiro que sai**, por isto este tipo de estrutura é chamada *LIFO* (Last In - First Out).

O exemplo mais prático que costuma utilizar-se para entender o processo de pilha é como uma pilha de livros ou pilha de pratos, no qual ao se colocar diversos elementos uns sobre os outros, se quisermos pegar o livro mais abaixo deveremos tirar todos os livros que estiverem sobre ele.



Uma pilha geralmente suporta as seguintes operações básicas:

- **Empilhar (push)** : insere um novo elemento no topo da pilha;
- **Desempilhar (pop)**: remove o elemento do topo da pilha;
- **Topo (top)**: acessa-se o elemento posicionado no topo da pilha;
- **Vazia (isEmpty)**: indica se a pilha está vazia.
- **Tamanho (size):** retorna a quantidade de elementos da pilha.

Exemplo de utilização de uma pilha:

Operação	**Pilha (topo) ----- (base)**	**Retorno**	**Tamanho da Pilha**
Empilhar(1)	1		1
Vazia	1	False	1
Empilhar(2)	2,1		2
Topo	2,1	2	2
Empilhar(7)	7,2,1		3
Tamanho	7,2,1	3	3
Desempilhar	2,1	7	2
Desempilhar	1	2	1
Desempilhar		1	0
Vazia		True	0

Implementação de Uma Pilha Estática

```
// Esta pilha armazena em cada posição da pilha um dado do tipo String.

// Pilha.cs
using System;
using System.Collections.Generic;
using System.Text;

namespace PilhaEstatica
{
    // definição da classe Pilha
    public class Pilha
    {
        private const int CAPACIDADE = 5; //define o tamanho maximo desta uma pilha.
        private string[] dados = new string[CAPACIDADE];   // vetor para guardar os dados da pilha.
        private int topo = -1; // variável que irá indicar a posição no vetor do topo da pilha.

        // este método informa o tamanho da pilha
        public int Tamanho()
        {
            return topo + 1; // lembre-se que o vetor inicia da posição zero...
        }

        // este método retorna true se a pilha estiver vazia
        public bool Vazia()
        {
            return Tamanho() == 0;
        }


        // este método empilha um valor string na pilha
        public void Empilha(string p_valor)
        {
            if (Tamanho() != CAPACIDADE)
            {
                topo++;
                dados[topo] = p_valor;
            }
            else
                throw new Exception("A pilha está cheia!!!");
        }

        // este método desempilha um valor da pilha
        public string Desempilha()
        {
            if (Vazia() == true)
                throw new Exception("A pilha está vazia!!!");
            else
            {
                string retorno = dados[topo];
                dados[topo] = null;
                topo--;
                return retorno;
            }
        }

        // este método devolve o valor que está no topo
        public string RetornaTopo()
        {
            if (Vazia() == true)
                throw new Exception("A pilha está vazia!!!");
            else
                return dados[topo];
        }
    }
}
```

```
// Programa principal (program.cs)

using System;
using System.Collections.Generic;
using System.Text;

namespace PilhaEstatica
{
    class Program
    {
        static void Main(string[] args)
        {
            int opcao = 0;
            string valor;
            Pilha minhaPilha = new Pilha(); // cria uma instância da classe pilha!

            do
            {
                try
                {
                    Console.Write("\n\n Escolha: 1-> empilha  2->desempilha  " +
                                  " 3->topo  4-> tamanho  9-> sair : ");
                    opcao = Convert.ToInt32(Console.ReadLine());

                    if (opcao == 1)
                    {
                        Console.Write(">>Digite o valor que deseja empilhar: ");
                        valor = Console.ReadLine();
                        minhaPilha.Empilha(valor);
                    }
                    else if (opcao == 2)
                    {
                        valor = minhaPilha.Desempilha();
                        Console.WriteLine(">>Desempilhado: {0} \n\n", valor);
                    }
                    else if (opcao == 3)
                    {
                        valor = minhaPilha.RetornaTopo();
                        Console.WriteLine(">>Valor no topo: {0} \n\n", valor);
                    }
                    else if (opcao == 4)
                    {
                        Console.WriteLine(">>Tamanho da pilha: {0}", minhaPilha.Tamanho());
                    }
                    else if (opcao == 9)
                    {
                        // sai do programa
                    }
                }
                catch (Exception erro)
                {
                    Console.WriteLine("ERRO: " + erro.Message);
                }
            }
            while (opcao != 9);
        }
    }
}
```

Exercícios sobre Pilhas

1. Crie uma pilha que manipule a seguinte estrutura:

Classe Funcionario
  Nome : string;
  Salario : Double;

Depois faça um programa para testar esta pilha (como no programa exemplo sobre pilhas).

A pilha deve possuir os seguintes métodos:

- Empilhar (Funcionario); ⇒ Empilhar um dado do tipo da estrutura que você definir.
- Desempilhar() : Funcionario; ⇒ Desempilhar um valor e retornar o valor desempilhado
- RetornaTopo() : Funcionario; ⇒ Retorna o valor que está no topo da pilha
- Tamanho() : int; ⇒ Retorna o tamanho da pilha

Observe que não há o método Vazio(). Portanto, para saber se a pilha está vazia, você deverá utilizar o método Tamanho().

No programa principal (onde você irá testar a pilha), adicione as seguintes opção ao menu do programa:
- Listar os dados da pilha
- Somar Salários

Observe que em ambos os métodos, você NÃO PODERÁ UTILIZAR O VETOR DA PILHA PARA SOLUCIONAR O PROBLEMA. Você tem a disposição apenas os métodos da pilha: Empilhar, Desempilhar, RetornaTopo e Tamanho.

2. Faça um método no programa principal que ao ser executado remove a base da pilha. Não altere nada na classe pilha.

3. Dada uma pilha P, execute:

```
P.empilha('a')
p.empilha('j')
p.empilha('h')
p.tamanho()
p.retornaTopo()
p.Desempilha()
p.empilha( p.retornaTopo())
p.empilha( p.desempilha())
```

6	
5	
4	
3	
2	
1	
0	
base	

FILAS

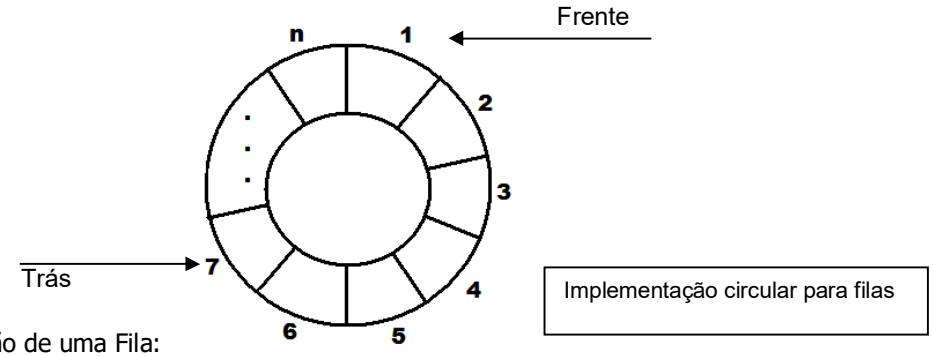
Uma Fila (*Queue*) é um caso particular de Listas Lineares que obedece ao critério FIFO (*First In, Firts Out*, ou Primeiro a Entrar, Primeiro a Sair). Numa fila os elementos são inseridos em uma das extremidades e retirados na outra extremidade. Existe uma ordem linear para a fila que é a "ordem de chegada". Filas são utilizadas quando desejamos processar itens de acordo com a ordem "primeiro-que-chega, primeiro-atendido".

Uma fila normalmente possui as seguintes operações (métodos):

- **Enfileira (valor)** ⇒ Insere o valor no final da fila.
- **Desenfileira** ⇒ Retorna o elemento do início da fila, desenfileirando-o.
- **Vazia** ⇒ Informa se a fila está fazia
- **Tamanho** ⇒ retorna o tamanho da fila
- **RetornaInicio** ⇒ retorna o elemento do início da fila, mas não o desenfileira.
- **RetornaFim** ⇒ retorna o elemento do final da fila, mas não o desenfileira.

Em uma implementação com arranjos (vetores), os itens são armazenados em posições contíguas da memória. Por causa das características da fila, a operação enfileira faz a parte de trás da fila expandir-se e a operação desenfileira faz a parte da frente da fila contrair-se. Conseqüentemente a fila tende a caminhar pela memória do computador, ocupando espaço na parte de trás e descartando espaço na parte da frente. Com poucas inserções e retiradas de itens, a fila vai ao encontro do limite do espaço de memória alocado para ela.

A solução para o problema acima é imaginar um vetor como um círculo, em que a primeira posição segue a última. Observe que a fila segue o sentido horário. Conforme os elementos vão sendo desenfileirados, a fila anda no sentido horário. O mesmo ocorre com os itens que vão sendo enfileirados. Para evitar sobrepor elementos no vetor, devemos verificar o tamanho da fila antes de efetuar a operação enfileirar.
Os elementos Frente e Trás são variáveis que indicarão em que posição no vetor estão o primeiro e o último elemento inserido na fila.



Implementação circular para filas

Exemplo de utilização de uma Fila:

Operação	Fila Fim ---→ Início	Retorno	Tamanho da Fila
Tamanho		0	0
Enfileira('A')	A		1
Enfileira('B')	BA		2
Vazia	BA	False	2
Enfileira('C')	CBA		3
RetornaInício	CBA	A	3
RetornaFim	CBA	C	3
Desenfilera	CB	A	2
Desenfilera	C	B	1
Desenfilera		C	0
Vazia		True	0
RetornaInicio		'ERRO'	0

Implementação da Fila Estática Circular

```
// Classe Fila.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FilaCircularEstatica
{
    class Fila
    {
        const int CAPACIDADE = 5; // capacidade máxima da fila
        private int quantidade = 0; // qtde de elementos enfileirados
        private int inicio = 0; // indica qual a primeira posição da fila
        private int fim = 0; // indica a próxima posição
        private string[] dados = new string[CAPACIDADE]; // armazenar os dados da fila

        // retorna o tamanho da fila
        public int Tamanho()
        {
            return quantidade;
        }

        // enfileira um valor string
        public void Enfileirar(string p_valor)
        {
            if (Tamanho() == CAPACIDADE)
            {
                throw new Exception("A fila está cheia!!!!");
            }
            else
            {
                dados[fim] = p_valor;
                fim = (fim + 1) % CAPACIDADE;
                quantidade++;
            }
        }

        // remove o primeiro elemento da fila e devolve.
        public string Desenfileira()
        {
            if (Tamanho() == 0)
            {
                throw new Exception("A fila está vazia!");
            }
            else
            {
                string valor = dados[inicio];
                inicio = (inicio + 1) % CAPACIDADE;
                quantidade--;
                return valor;
            }
        }
    }
}
```

```
// Programa principal (Program.cs)
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FilaCircularEstatica
{
    class Program
    {
        static void Main(string[] args)
        {
            string opcao = "", valor;
            Fila minhafila = new Fila();
            Console.WriteLine("Sistema em C# para testar a execução de uma fila circular\n");
            do
            {
                try
                {
                    Console.WriteLine("\n\nDigite: 1->Enfileirar 2->Desenfileirar " +
                    "3-> Tamanho 9->Sair");
                    opcao = Console.ReadLine();
                    switch (opcao)
                    {
                        case "1":
                            Console.WriteLine("Digite um valor para enfileirar:");
                            valor = Console.ReadLine();
                            minhafila.Enfileirar(valor);
                            break;
                        case "2":
                            Console.WriteLine("Desenfileirado: {0}", minhafila.Desenfileira());
                            break;
                        case "3":
                            Console.WriteLine("Tamanho da fila:{0}", minhafila.Tamanho());
                            break;
                        case "9":
                            Console.WriteLine("Saindo do sistema...");
                            break;
                        default:
                            Console.WriteLine("Opção inválida!!!");
                            break;
                    }
                }
                catch (Exception erro)
                {
                    Console.WriteLine(erro.Message);
                }
            } while (opcao != "9");
        }
    }
}
```

Exercício de Fila e Pilha:

1-) Dada uma Fila Circular F e uma pilha P, execute os métodos abaixo e preencha as estruturas pilha e fila.

Fila Circular - F
INICIO

0	1	2	3	4

Pilha - P
BASE

0	1	2	3	4

P.empilha('a')
P.empilha('b')
P.empilha('c')
P.empilha(P.retornaTopo)
F.enfileira('F')
F.enfileira(P.retornaTopo)
F.enfileira(P.desempilha)
F.enfileira(P.retornaTopo)
P.empilha(F.desenfileira)
P.empilha(P.desempilha)
F.enfileira(F.desenfileira)

2-) Implemente na fila circular os métodos RetornaInicio e RetornaFim. Ambos os métodos devem apenas retornar o conteúdo (string) da fila. Altere o programa principal para exibir os dados retornados.

3-) Implemente na fila circular o método Listar, cujo objetivo é exibir em vídeo os elementos da fila, onde os elementos à esquerda do vídeo representam os valores iniciais da fila.

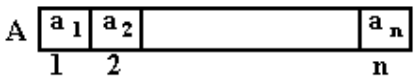
4-) Faça um programa para inverter os dados de uma Fila. Utilize a classe fila que foi passada como exemplo (string). Não altere a classe Fila, a inversão deve ser feita no programa principal (program.cs). Você deve utilizar apenas os métodos públicos da Fila. Tente utilizar uma pílha para solucionar este problema.

LISTAS ESTÁTICAS

http://www.inf.ufsc.br/~ine5384-hp/Capitulo2/EstruturasLista.html
http://www.icmc.usp.br/~sce182/lestse.html

Uma lista estática sequencial é um arranjo de registros onde estão estabelecidas regras de precedência entre seus elementos ou é uma coleção ordenada de componentes do mesmo tipo. O sucessor de um elemento ocupa posição física subsequente.

Ex: lista telefônica, lista de alunos



- Este conjunto de dados pode possuir uma ordem intrínseca (Lista Ordenada) ou não;
- Este conjunto de dados deve ocupar espaços de memória fisicamente consecutivos (implementado com vetor);
- Se os dados estiverem dispersos fisicamente na memória, para que este conjunto seja uma lista, ele deve possuir operações e informações adicionais que permitam que seja tratado como tal (Lista Encadeada).

O conjunto de operações a ser definido depende de cada aplicação. Um conjunto de operações necessário a uma maioria de aplicações é:

- 1. Criar uma lista linear vazia.
- 2. Inserir um novo item imediatamente após o i-ésimo item.
- 3. Retirar o i-ésimo item.
- 4. Localizar o i-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.
- 5. Combinar duas ou mais listas em uma lista única.
- 6. Partir uma lista em duas ou mais listas.
- 7. Fazer uma cópia da lista.
- 8. Ordenar os itens da lista em ordem ascendente ou descendente, de acordo com alguns de seus componentes.
- 9. Pesquisar a ocorrência de um item com um valor particular em algum componente.

DETAHES DA IMPLEMENTAÇÃO DE LISTAS ESTÁTICAS POR MEIO DE ARRANJOS

- Os itens da lista são armazenados em posições contíguas de memória.
- A lista pode ser percorrida em qualquer direção.
- A inserção de um novo item pode ser realizada após o último item com custo constante.
- A inserção de um novo item no meio da lista requer um deslocamento de todos os itens localizados após o ponto de inserção.
- Retirar um item do início da lista requer um deslocamento de itens para preencher o espaço deixado vazio.

Vantagem:

- Acesso direto indexado a qualquer elemento da lista
- Tempo constante para acessar o elemento i - dependerá somente do índice.

Desvantagem:

- Movimentação quando eliminado/inserido elemento
- Tamanho máximo pré-determinado

Quando usar:

- Listas pequenas
- Inserção/remoção no fim da lista
- Tamanho máximo bem definido

Implementação de Lista Estática

Classe Lista.cs

```
class Lista
{
    private const int CAPACIDADE = 10;
    private string[] dados = new string[CAPACIDADE];
    private int quantidade = 0;

    public int Tamanho()
    {
        return quantidade;
    }

    public void InsereNaPosicao(int p_posicao, string p_valor)
    {
        if (Tamanho() == CAPACIDADE)
            throw new Exception("A lista está cheia!!!\n\n");
        else if (p_posicao > Tamanho())
            throw new Exception("Não é possível inserir nesta posição");
        else
        {
            quantidade++;
            for (int i = Tamanho() - 1; i > p_posicao; i--)
            {
                dados[i] = dados[i - 1];
            }
            dados[p_posicao] = p_valor;
        }
    }


    public string RemoveDaPosicao(int posicao)
    {
        if (Tamanho() == 0)
            throw new Exception("A lista está vazia!!!");
        else if (posicao > Tamanho() -1)
            throw new Exception("Posição inválida!!!");
        else
        {
            string aux = dados[posicao];
            for (int i = posicao; i < Tamanho() - 1; i++)
            {
                dados[i] = dados[i + 1];
            }
            quantidade--;
            return aux;
        }
    }


    public void InsereNoInicio(string p_valor)
    {
        InsereNaPosicao(0, p_valor);
    }


    public void InsereNoFim(string p_valor)
    {
        InsereNaPosicao(Tamanho(), p_valor);
    }

    public void ImprimeLista()
    {
        Console.WriteLine("\n\nImpressão dos dados da lista:\n");
        for (int i = 0; i < Tamanho(); i++)
        {
            Console.WriteLine(dados[i]);
        }
    }
}
```

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        string opcao = "", valor;
        int posicao;
        Lista minhaLista = new Lista();
        Console.WriteLine("Sistema em C# para testar a execução de uma lista estática\n");
        do
        {
            try
            {
                Console.WriteLine("\nDigite: \n 1-> Inserir no início \n " +
                "2-> Inserir no fim \n " +
                "3-> Inserir em uma posição (lembre-se que inicia do do zero!)\n " +
                "4-> Tamanho \n 5-> Listar \n " +
                "6-> Remover elemento de uma posição \n 9-> Sair");
                opcao = Console.ReadLine();
                switch (opcao)
                {
                    case "1":
                        Console.WriteLine("Digite um valor para inserir no início:");
                        valor = Console.ReadLine();
                        minhaLista.InsereNoInicio(valor);
                        break;
                    case "2":
                        Console.WriteLine("Digite um valor para inserir no fim:");
                        valor = Console.ReadLine();
                        minhaLista.InsereNoFim(valor);
                        break;
                    case "3":
                        Console.WriteLine("Digite um valor para inserir:");
                        valor = Console.ReadLine();
                        Console.WriteLine("Digite a posição:");
                        posicao = Convert.ToInt32(Console.ReadLine());
                        minhaLista.InsereNaPosicao(posicao, valor);
                        break;
                    case "4":
                        Console.WriteLine("Tamanho da lista:{0}", minhaLista.Tamanho());
                        break;
                    case "5":
                        minhaLista.ImprimeLista();
                        break;
                    case "6":
                        Console.WriteLine("Digite a posição que deseja remover:");
                        posicao = Convert.ToInt32(Console.ReadLine());
                        Console.WriteLine("Removido:{0}", minhaLista.RemoveDaPosicao(posicao));
                        break;
                    case "9":
                        Console.WriteLine("Saindo do sistema...");
                        break;
                    default:
                        Console.WriteLine("Opção inválida!!!");
                        break;
                }
            }
            catch (Exception erro)
            {
                Console.WriteLine(erro.Message);
            }
        } while (opcao != "9");

    }
}
```

Exercícios Sobre Listas, Filas e Pilhas

Exercício 1:

Utilizando uma Lista (como a implementada na explicação sobre Listas) pode-se simular uma Pilha e uma Fila. Isso pode ser feito sem que seja necessário alterar nada na classe Lista.

Partindo deste princípio, crie uma classe Pilha que deverá ser implementada usando para isso uma Lista. Ou seja, na sua classe Pilha, não haverá mais um vetor DADOS para armazenar os elementos da pilha. No lugar disso, eles deverão ser armazenados em uma Lista.

A sua pilha deverá executar os métodos **Empilhar, Desempilhar e Tamanho.**

Utilize a Lista que foi explicada em aula (ela armazena apenas strings).

Exercício 2:

Faça a mesma coisa que no exercício anterior, porém, crie desta vez uma Fila.

Deve executar os métodos: **Enfileira, Desenfileira e Tamanho.**

Exercício 3:

Implementar uma lista que seja capaz de armazenar alunos.
Cada aluno possui as propriedades:

- RA
- Nome

O usuário poderá inserir nesta lista até 32 alunos.

Deverá ser implementado na classe lista os métodos RetornaPrimeiro e RetornaUltimo, que irão retornar, respectivamente, o primeiro e último elemento da lista.

Também deve haver um método para localizar um elemento na lista (pesquisar pelo RA).

Crie uma interface gráfica para cadastrar, remover e pesquisar alunos na lista.
Também deve ser possível testar os métodos RetornaPrimeiro e RetornaUltimo.
Deve ser possível remover um RA da lista.

Não permite RA´s repetidos na lista!!!

Exercício 4
Dada a lista estática abaixo, execute:

INICIO

0	1	2	3	4

InsereNoInicio("A");
InsereNoFim("Z");
InsereNaPosição (2, "X");
InsereNaPosição (0, "B");
InsereNaPosicao (1, "A");
RemoveDaposicao(0)
InsereNoInicio(RetornaPosicao (0));
InsereNoFim(removedaPosicao(2));

PONTEIROS (OU APONTADORES)

Ótimo material sobre apontadores:
http://br.geocities.com/cesarakg/pointers.html
http://www.deei.fct.ualg.pt/IC/t20_p.html

Um apontador ou ponteiro é um tipo de variável especial, cujo objetivo é armazenar um endereço da memória. Ou seja, ele não armazena um valor como "Olá mundo" ou 55. Ao invés disso, ele armazena um endereço na memória e, neste endereço, encontra-se uma informação útil, como um texto ou um número.

Um **apontador** contém o endereço de um lugar na memória.

Quando você executa comandos tais como:

`I = 10` ou `J = 1`

você está acessando o conteúdo da variável. O compilador procura automaticamente o endereço da variável e acessa o seu conteúdo. **Um apontador, entretanto, lhe permite determinar por si próprio o endereço da variável**.

Ex:

- I = 42;
- PonteiroParaI = &I;

Endereço	Conteúdo
E456	**E123**
E455	
E454	
•	
•	
•	
E123	**42**

Variável apontador: **PonteiroParaI**

Variável Inteira **I**:

Para mostrar o conteúdo do endereço para o qual a o ponteiro aponta, podemos utilizar:

```
Console.WriteLine( *PonteiroParaI );   { exibirá 42 }
```

&X retorna o endereço da variável **x** *p é o conteúdo do endereço **p**

Para que o apontador saiba exatamente o tamanho da informação para a qual ele aponta na memória (veja figura abaixo), uma variável do tipo apontador deve estar associada a um tipo específico de variável ou registro.

Para criar uma variável do tipo apontador, coloque o símbolo * na frente do tipo da variável

ex: `int* p1;`

```
          p
          ↓
..00100111100000011100011011101010001101000000..

..00100111100000011100011011101010001101000000..

..00100111100000011100011011101010001101000000..
```

Cada tipo de dado ocupa um tamanho diferente na memória. O apontador precisa saber para qual tipo de dado ele vai apontar, para que ele possa acessar corretamente a informação.

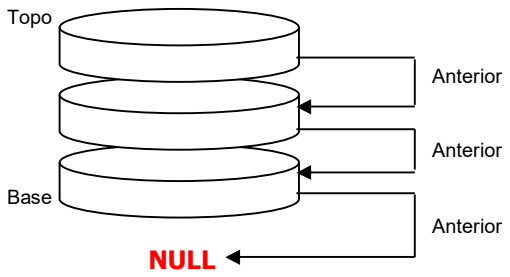
A esquerda temos 3 tipos de dados em Pascal: Byte na primeira linha, word na segunda e longint na terceira. Veja que os tamanhos são diferentes!

Exemplo de um programa que utiliza apontadores:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ponteiros
{
    class ProgramPonteiro
    {
        static void Main(string[] args)
        {
            // para rodar este programa, você deve configurar o seu projeto para rodar código não protegido.
            // para tanto, vá ao menu project-> properties (ultima opção do menu) -> escolha a aba BUILD
            // e marque a opção "ALLOW UNSAFE CODE". Salve o projeto e compile-o com F5.
            unsafe
            {
                int* p1;    // cria uma variável que pode apontar para uma
                            //outra variável inteira
                int numero = 7;

                // &numero = endereço da variável numero
                p1 = &numero; // o ponteiro p1  vai apontar para o mesmo endereço
                              // que a variável numero

                // *p1 -> valor armazenado no endereço apontado por p1
                Console.WriteLine( "Variável número: {0}  ponteiro:  {1}", numero, *p1);

                Console.Write("\nDigite um valor para o ponteiro:");
                *p1 = Convert.ToInt32(Console.ReadLine());

                Console.WriteLine("Variável número: {0} ",  numero);

                Console.ReadLine();
            }
        }
    }
}
```

PILHAS DINÂMICAS (UTILIZANDO ENCADEAMENTO)

As pilhas criadas utilizando-se apontadores são mais eficientes, pois não precisamos definir um Limite. Cada elemento da pilha aponta para o anterior, utilizando para isso apontadores.



Cada elemento da pilha possui:

- Valor que deve ser empilhado.
- Apontador para o **elemento anterior** na pilha.

A base da pilha tem como elemento anterior o valor **NULL.**

Para criar uma pilha utilizando apontadores, precisamos definir uma classe que possua os seguintes campos:

CLASSE NODO
Conteúdo do nodo (valor)
Endereço do nodo anterior

Implementação da Pilha dinâmica utilizando objetos encadeados

Classe Nodo.cs

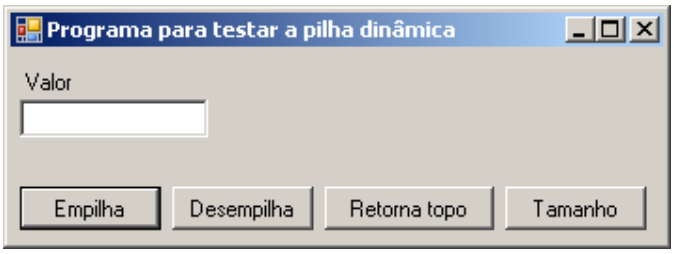
```
/// <summary>
/// Classe que irá representar 1 elemento na pilha
/// </summary>
class Nodo
{
    /// <summary>
    /// Valor que será armazenado
    /// </summary>
    public string Valor {get;set;}

    /// <summary>
    /// Endereço do nodo anterior na pilha
    /// </summary>
    public Nodo Anterior  {get;set;}
}
```

Classe Pilha.cs

```
/// <summary>
/// Classe Pilha Dinâmica
/// </summary>
class Pilha
{
    //Representa o topo da pilha
    private Nodo topo = null;

    // quantidade de elementos na pilha
    int quantidade = 0;
    public int Quantidade
    {
        get { return quantidade; }
    }

    /// <summary>
    /// Método para empilhar strings
    /// </summary>
    /// <param name="valor"></param>
    public void Empilhar(string valor)
    {
        Nodo novoNodo = new Nodo();
        novoNodo.Valor = valor;
        novoNodo.Anterior = topo;

        topo = novoNodo;
        quantidade++;
    }


    /// <summary>
    /// Desempilhar elementos da pilha
    /// </summary>
    /// <returns></returns>
    public string Desempilhar()
    {
        if (quantidade == 0)
            throw new Exception("A pilha está vazia!");
        else
        {
```

```
            string retorno = topo.Valor;
            topo = topo.Anterior;
            quantidade--;
            return retorno;
        }
    }

    /// <summary>
    /// Método para retornar o topo da pilha
    /// </summary>
    /// <returns></returns>
    public string RetornaTopo()
    {
        if (quantidade == 0)
            throw new Exception("A pilha está vazia!");
        else
        {
            return topo.Valor;
        }
    }
}
```

Formulário Principal para testar a pilha:




```
public partial class Form1 : Form
{
    Pilha pilha = new Pilha();

    public Form1()
    {
        InitializeComponent();
    }

    private void btnEmpilhar_Click(object sender, EventArgs e)
    {
        pilha.Empilhar(txtValor.Text);
        txtValor.Clear();
    }


    private void btnDesempilhar_Click(object sender, EventArgs e)
    {
        try
        {
            txtValor.Text = pilha.Desempilhar();
        }
        catch (Exception erro)
        {
            MessageBox.Show(erro.Message);
        }

    }

    private void btnTamanho_Click(object sender, EventArgs e)
```

```
        {
            MessageBox.Show(pilha.Quantidade.ToString());
        }

        private void btnRetornaTopo_Click(object sender, EventArgs e)
        {
            try
            {
                txtValor.Text = pilha.RetornaTopo();
            }
            catch (Exception erro)
            {
                MessageBox.Show(erro.Message);
            }
        }
    }
```

Exercícios:

1-) Implemente na classe da pilha dinâmica um método para retornar um string com todos os elementos empilhados. Você pode separar os elementos com um "-".

2-) Implemente a Fila utilizando a mesma técnica utilizada na pilha dinâmica. Observe que não será uma Fila circular, pois não há mais o problema de ter que descolar os elementos quando utilizamos esta técnica de elementos encadeados.

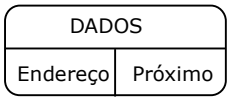
3-) Altere a classe Fila dinâmica para que ela armazene objetos da classe Alunos (RA int, nome string).

4-) Implemente na classe Fila um método para listar os seus elementos. Devolva um string. Você pode separar os elementos com um "-".

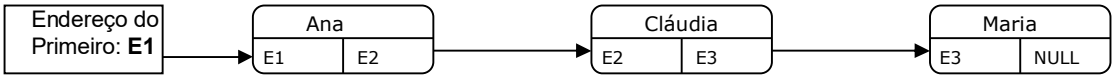
LISTAS SIMPLESMENTE ENCADEADAS

Material retirado da referência [2], [3] e [4].

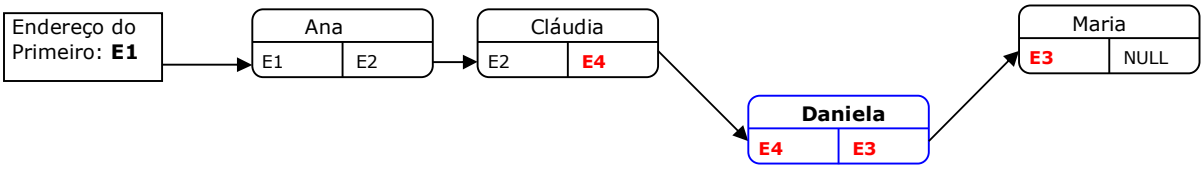
Em uma lista simplesmente encadeada, cada elemento contém um apontador que aponta para a o elemento seguinte. Na implementação de listas utilizando vetores, os dados ocupavam posições contíguas da memória. Sendo assim, sempre que incluímos ou apagamos um elemento no meio da lista, precisamos reorganizar os dados do vetor, o que computacionalmente pode ser muito custoso. Nas listas simplesmente encadeadas, os dados não ocupam posições contíguas da memória, portando operações de remoção e inclusão são executadas muito mais rapidamente. Um elemento de uma lista simplesmente encadeada pode ser definido como na figura abaixo:



Quando criamos uma lista utilizando apontadores, precisamos ter uma variável que aponta sempre para o início da lista. Abaixo, temos um exemplo de uma lista simplesmente encadeada para armazenar nomes em ordem alfabética:

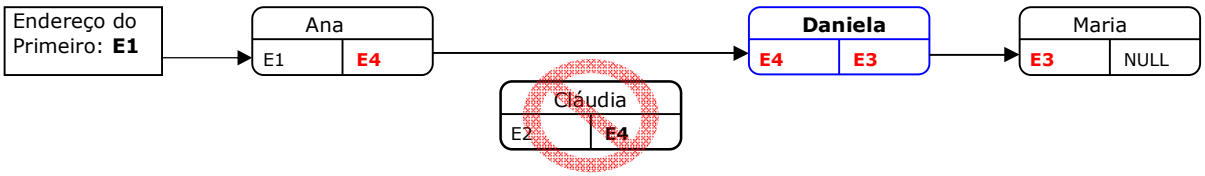


Para **incluir** um novo elemento, por exemplo, o nome Daniela, devemos apenas alterar o apontador <u>próximo</u> do elemento que está no endereço E2. Veja abaixo:



<u>Observe que a ordem dos endereços não importa. O que importa é a ordem que eles estão encadeados!</u>

O mesmo ocorre ao se **remover** um elemento da lista. Veja abaixo como ficaria a remoção do elemento Cláudia:



Código fonte da lista dinâmica simplesmente encadeada

```
public class Nodo
{
    public string Dado {get;set;}
    public Nodo Proximo { get; set;}

    /// <summary>
    /// Construtor sem parâmetros
    /// </summary>
    public Nodo()
    {
        Dado = string.Empty;
        Proximo = null;
    }
}
```

```
public class Lista
{
    Nodo primeiro = null; // ponteiro para o primeiro elemento da lista
    int qtde = 0;


    public int Quantidade
    {
        get { return qtde; }
    }

    /// <summary>
    /// Método para inserir um valor na lista
    /// </summary>
    /// <param name="anterior">o nodo que será o anterior ao nodo inserido.
    /// Se o novo nodo for o primeiro, passe null</param>
    /// <param name="valor">o valor a ser inserido</param>
    private void InserirNaPosicao(Nodo anterior, string valor)
    {
        Nodo novo = new Nodo();
        novo.Dado = valor;
        qtde++;

        if (anterior == null) //indica que será o primeiro da lista
        {
            novo.Proximo = primeiro;
            primeiro = novo;
        }
        else
        {
            novo.Proximo = anterior.Proximo;
            anterior.Proximo = novo;
        }
    }

    /// <summary>
    /// Insere um valor no início da lista
    /// </summary>
    /// <param name="valor"></param>
    public void InserirNoInicio(string valor)
    {
        InserirNaPosicao(null, valor);
    }
```

```
        /// <summary>
        /// Insere um valor no final da lista
        /// </summary>
        /// <param name="valor"></param>
        public void InserirNoFim(string valor)
        {
            if (qtde == 0)
                InserirNoInicio(valor);
            else
            {
                Nodo aux = primeiro;
                while (aux.Proximo != null)
                    aux = aux.Proximo;
                InserirNaPosicao(aux, valor);
            }
        }

        /// <summary>
        /// Insere em uma posição, iniciando do 0
        /// </summary>
        /// <param name="valor">valor</param>
        /// <param name="posicao">posicao iniciando do 1</param>
        public void InserirNaPosicao(string valor, int posicao)
        {
            if (posicao > qtde || posicao < 0)
                throw new Exception("Não é possível inserir.");
            if (posicao == 0)
                InserirNoInicio(valor);
            else
            {
                //descobre qual é o nodo anterior ao que será incluído
                Nodo aux = primeiro;
                for (int i = 1; i < posicao; i++)
                    aux = aux.Proximo;
                InserirNaPosicao(aux, valor);
            }
        }

        /// <summary>
        /// Remove um elemento da lista com base em sua posição, que inicia
        /// do zero
        /// </summary>
        /// <param name="posicao">posição</param>
        public string RemoverDaPosicao(int posicao)
        {
            string retorno = "";
            if (posicao >= qtde || posicao < 0 || qtde == 0)
                throw new Exception("Não é possível remover.");
            if (posicao == 0)
                primeiro = primeiro.Proximo;
            else
            {
                //descobre qual é o nodo anterior que será excluido
                Nodo aux = primeiro;
                for (int i = 1; i < posicao; i++)
                    aux = aux.Proximo;

                retorno = aux.Proximo.Dado;

                aux.Proximo = aux.Proximo.Proximo;
            }
            qtde--;

            return retorno;
```

```
        }


        /// <summary>
        /// Retorna um string com todos os elementos da lista concatenados
        /// </summary>
        /// <returns></returns>
        public string Listar()
        {
            string r = string.Empty;
            Nodo aux = primeiro;
            while (aux != null)
            {
                r = r + Environment.NewLine + aux.Dado;
                aux = aux.Proximo;
            }
            return r.Trim();
        }
    }
```

Observe que a lista acima não possui um ponteiro (referência) para o último elemento da lista. Se tivesse, as operações de inserir no final da lista seriam O(1) no lugar de O(n).

Programa principal para teste:

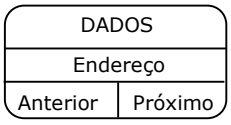
```
class Program
    {
        static void Main(string[] args)
        {
            string opcao = "", valor;
            int posicao;
            Lista minhaLista = new Lista();
            Console.WriteLine("Sistema em C# para testar a execução de uma lista estática\n");
            do
            {
                try
                {
                    Console.WriteLine("\nDigite: \n 1-> Inserir no início \n " +
                    "2-> Inserir no fim \n " +
                    "3-> Inserir em uma posição (lembre-se que inicia do do zero!)\n " +
                    "4-> Tamanho \n 5-> Listar \n " +
                    "6-> Remover elemento de uma posição \n 9-> Sair");
                    opcao = Console.ReadLine();
                    switch (opcao)
                    {
                        case "1":
                            Console.WriteLine("Digite um valor para inserir no início:");
                            valor = Console.ReadLine();
                            minhaLista.InserirNoInicio(valor);
                            break;
                        case "2":
                            Console.WriteLine("Digite um valor para inserir no fim:");
                            valor = Console.ReadLine();
                            minhaLista.InserirNoFim(valor);
                            break;
                        case "3":
                            Console.WriteLine("Digite um valor para inserir:");
```

```
                        valor = Console.ReadLine();
                        Console.WriteLine("Digite a posição:");
                        posicao = Convert.ToInt32(Console.ReadLine());
                        minhaLista.InserirNaPosicao(valor, posicao);
                        break;
                    case "4":
                        Console.WriteLine("Tamanho da lista:{0}", minhaLista.Quantidade);
                        break;
                    case "5":
                        Console.WriteLine( minhaLista.Listar());
                        break;
                    case "6":
                        Console.WriteLine("Digite a posição que deseja remover:");
                        posicao = Convert.ToInt32(Console.ReadLine());
                        Console.WriteLine("Removido:{0}", minhaLista.RemoverDaPosicao(posicao)
);
                        break;
                    case "9":
                        Console.WriteLine("Saindo do sistema...");
                        break;
                    default:
                        Console.WriteLine("Opção inválida!!!");
                        break;
                }
            }
            catch (Exception erro)
            {
                Console.WriteLine(erro.Message);
            }
        } while (opcao != "9");
    }
}
```

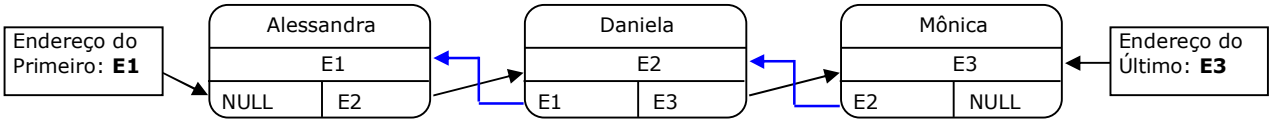
LISTAS DUPLAMENTE ENCADEADAS

Material retirado da referência [2], [3] e [4].

A diferença de uma lista duplamente encadeada para uma lista simplesmente encadeada é que em uma lista duplamente encadeada cada elemento contém um segundo apontador que aponta para o elemento que o antecede. Assim, você não precisa se preocupar mais com o início da lista. Se você tiver um apontador para qualquer elemento da lista, pode encontrar o caminho para todos os outros elementos. Em uma lista duplamente encadeada, são necessárias variáveis para apontar para o início e para o final da lista. Abaixo temos a representação de um elemento de uma lista duplamente encadeada:



Exemplo de uma lista duplamente encadeada para armazenar nomes em ordem alfabética:



Para **Inserir** e **Remover** elementos, o processo é semelhante ao apresentado na lista simplesmente encadeada. A diferença é que na lista duplamente encadeada é necessário também atualizar o campo "anterior" dos elementos.

Exercício de LISTA

Exercício 1: Crie uma lista duplamente encadeada para guardar os seguintes nomes de cidades em ordem alfabética:

- Mauá
- São Bernardo do campo
- Diadema (é cidade?)
- Ribeirão pires

Remova da lista acima Mauá.
Adicione na lista agora as cidades: Mirassol , sobradinho e Cajamar.
Começar os endereços do E1

Exercício 2: Na lista simplesmente encadeada, implemente o método para pesquisar um elemento na lista. Pesquise pelo dado e retorne true se existir ou false caso contrário.

Exercício 3: Implemente a classe Lista duplamente encadeada, incluindo, além dos métodos que foram expostos na lista simplesmente encadeada, os métodos para retornar o primeiro e o último elemento da lista e os métodos para inserir no início e inserir no final da lista. A lista deverá guardar em seu conteúdo um string.

RECURSIVIDADE OU RECURSÃO

Material retirado de:
http://pt.wikipedia.org/wiki/Recursividade (muito bom)
http://pt.wikipedia.org/wiki/Recursividade_%28ci%C3%AAncia_da_computa%C3%A7%C3%A3o%29 (ótimo)
http://www.di.ufpe.br/~if096/recursao/ (bom)
Referência [1]

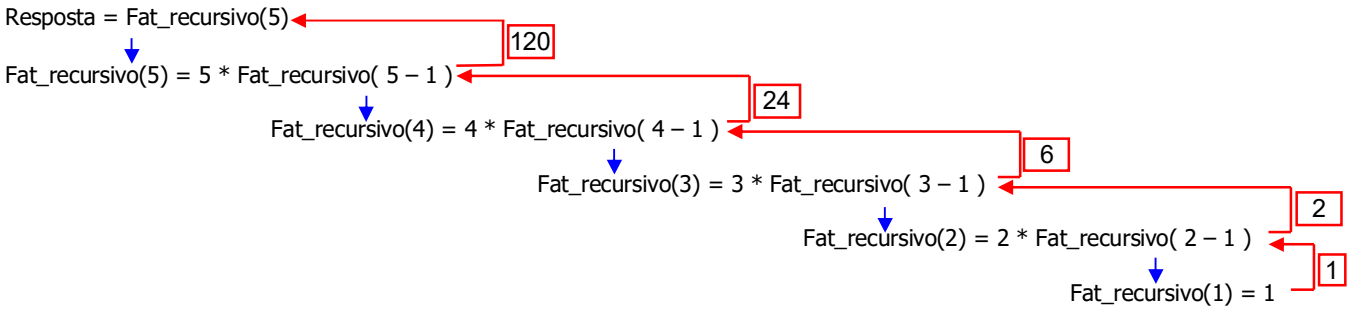
O que é Recursividade: Uma Rotina ou Função é recursiva quando ela chama a si mesma, seja de forma direta ou indireta. Uma função recursiva DEVE ter um ponto de parada, ou seja, em algum momento ela deve parar de se chamar. Praticamente todas as linguagens oferecem suporte a recursividade. Trata-se de uma técnica de programação.

Por exemplo, segue uma definição recursiva da ancestralidade de uma pessoa:
- Os pais de uma pessoa são seus antepassados (caso base);
- Os pais de qualquer antepassado são também antepassados da pessoa em consideração (passo recursivo).

Cálculo do Fatorial **sem Recursão**, usamos apenas uma estrutura de repetição (iterador)	Cálculo do Fatorial **com Recursão** direta

`long fat_iterativo(int numero)`
`{`
`  long r=1;`
 	
`  for (int i=2; i<= numero; i++)`
`  {`
`  r = r * i;`
`  }`
`  return r;`
`}`
	`long fat_recursivo(int numero)`
	`{`
	`  if (numero == 0)`
	`  return 1;`
	`  else if (numero >= 2)`
	`  return numero*fat_recursivo(numero-1);`
	`  else`
	`  return numero;`
	`}`

Teste de Mesa do Fatorial de 5: azul = ida(chamada recursiva) , vermelho = volta (retorno da função recursiva)



Recursão versus Iteração

No exemplo do fatorial, a implementação iterativa tende a ser ligeiramente mais rápida na prática do que a implementação recursiva, uma vez que uma implementação recursiva precisa registrar o estado atual do processamento de maneira que ela possa continuar de onde parou após a conclusão de cada nova excecução subordinada do procedimento recursivo. Esta ação consome tempo e memória.
Existem outros tipos de problemas cujas soluções são inerentemente recursivas, já que elas precisam manter registros de estados anteriores. Um exemplo é o percurso de uma árvore;

Toda função que puder ser produzida por um computador pode ser escrita como função recursiva sem o uso de iteração; reciprocamente, qualquer função recursiva pode ser descrita através de iterações sucessivas. Todos altoritmo recursivo pode ser implementado iterativamente com a ajuda de uma pilha, mas o uso de uma pilha, de certa forma, anula as vantagens das soluções iterativas.

Tipos de Recursividade:

Direta: Quando chama a si mesma, quando dada situação requer uma chamada da própria Rotina em execução para si mesma. Ex: O exemplo de fatorial recursivo dado acima.

Indireta: Funções podem ser recursivas (invocar a si próprias) indiretamente, fazendo isto através de outras funções: assim, "P" pode chamar "Q" que chama "R" e assim por diante, até que "P" seja novamente invocada.

Ex:

```
double Calculo( double a,b )
{
   return Divide(a,b) + a + b;
}
```

```
double Divide( double a, b )
{
   if (b == 0)
       b  = Calculo(a, b + a);

   return a/ b;
}
```

Aqui ocorre a recursividade indireta!

Note que a função `fatorial` usada como exemplo na seção anterior *não* é recursiva em cauda, pois depois que ela recebe o resultado da chamada recursiva, ela deve multiplicar o resultado por `VALOR` antes de retornar para o ponto em que ocorre a chamada.

Qual a desvantagem da Recursão?
Cada chamada recursiva implica em maior tempo e espaço, pois, toda vez que uma Rotina é chamada, todas as variáveis locais são recriadas.

Qual a vantagem da Recursão?
Se bem utilizada, pode tornar o algoritmo: elegante, claro, conciso e simples. Mas, é preciso antes decidir sobre o uso da Recursão ou da Iteração.

Exercícios

Exercício 1:

Faça um programa para calcular a potencia de um número. O método recursivo deve receber como parâmetro a base e o expoente, e devolver o valor da potência.

EX: double CalculaPotencia (int p_base, int p_expoente)
CalculaPotencia (2,3) = 8

Exercício 2:

Faça o teste de mesa do seu método para os valores informados acima.

Exercício 3:

Faça um programa para imprimir a tabuada, usando recursividade.

Exercício 4:

Faça um método para limpar todos os makedit e os Textbox de um form, mesmo que eles estejam dentro de panels, groupbox.

Exercício 5:

Progressão aritmética é um tipo de sequência numérica que a partir do segundo elemento cada termo (elemento) é a soma do seu antecessor por uma constante.

(5,7,9,11,13,15,17) essa sequência é uma Progressão aritmética, pois os seus elementos são formados pela soma do seu antecessor com a constante 2.
a1 = 5
a2 = 5 + 2 = 7
a3 = 7 + 2 = 9
a4 = 9 + 2 = 11
a5 = 11 + 2 = 13
a6 = 13 + 2 = 15
a7 = 15 + 2 = 17

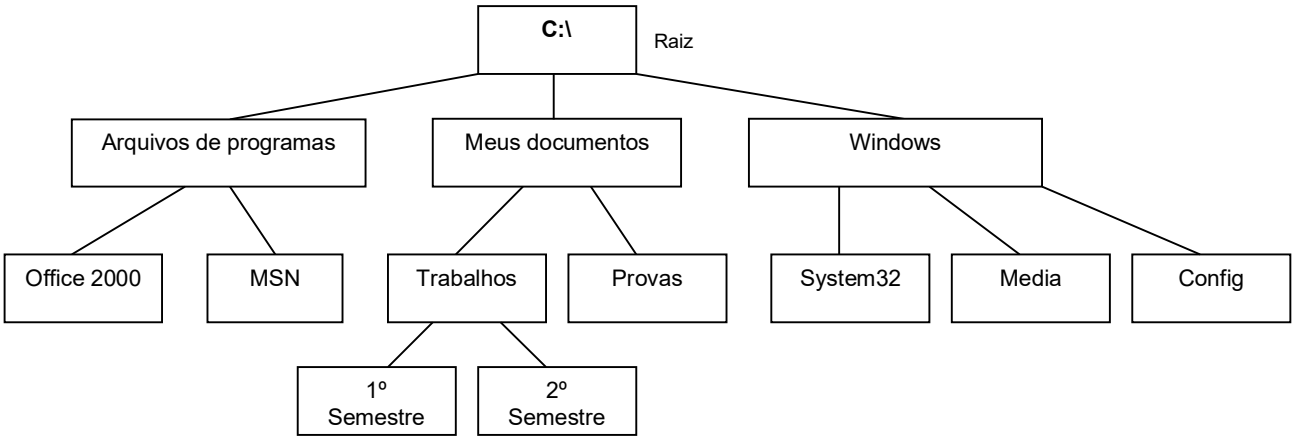
Faça um programa que solicite: O elemento inicial da PA, a constante (razão) e o total de sequencias que se deseja gerar e então faça 2 algoritmos para resolver este problema: 1 recursivo e um sem recursividade.

ÁRVORES

Material retirado da referência [3].

Uma **árvore** é um tipo abstrato de dados que armazena elementos de maneira hierárquica. Como exceção do elemento do topo, cada elemento tem um elemento pai e zero ou mais elementos filhos. Uma árvore normalmente é desenhada colocando-se os elementos dentro de elipses ou retângulos e conectando pais e filhos com linhas retas. Normalmente o elemento topo é chamado de raiz da árvore, mas é desenhado como sendo o elemento mais alto, com todos os demais conectados abaixo (exatamente ao contrário de uma árvore real).

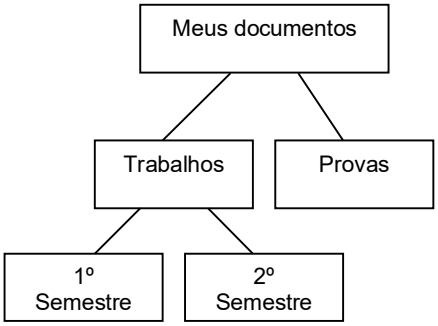
EX: Uma árvore que representa a estrutura de pastas em um Hard Disk:



Uma árvore **T** é um conjunto de **nodos** que armazenam elementos em relacionamentos **pai-filho** com as seguintes propriedades:

- **T** tem um nodo especial, **r**, chamado de **raiz** de **T**.
- Cada nodo **v** de **T** diferente de **r** tem um nodo pai **u**.
- Se um nodo **u** é pai de um nodo **v**, então dizemos que **v** é filho de **u**.
- Dois nodos que são filhos de um mesmo pai são **irmãos**.
- Um nodo é **externo (ou folha)** se não tem filhos.
- Um nodo é **interno** se tem um ou mais filhos.
- Um **subárvore** de **T** enraizada no nodo **v** é a árvore formada por todos os descendentes de **v** em **T** (incluindo o próprio **v**).
- O **ancestral** de um nodo é tanto um ancestral direto como um ancestral do pai do nodo.
- Um nodo **v** é **descendente** de **u** se **u** é um ancestral de **v**. Ex: na figura acima, Meus documentos é ancestral de 2º Semestre e 2º Semestre é descendente de Meus documentos.
- Seja **v** um nodo de uma árvore **T**. A **profundidade** de **v** é o número de ancestrais de **v**, excluindo o próprio **v**. Observe que esta definição implica que a profundidade da raiz de **T** é 0 (zero). Como exemplo, na figura acima, a profundidade do nodo Trabalhos é 2, e a profundidade do nodo 2º semestre é 3.
- A **altura** de um nodo é o comprimento do caminho mais longo desde nodo até um nó folha ou externo. Sendo assim, a altura de uma árvore é a altura do nodo Raiz. No exemplo acima, a árvore tem altura 3. Também se diz que a altura de uma árvore **T** é igual à profundidade máxima de um nodo externo de **T**.

A figura abaixo representa uma subárvore da árvore acima. Esta subárvore possui 5 nodos, onde 2 são nodos internos (Meus Documentos e Trabalhos) e 3 são nodos externos (Provas, 1º Semestre e 2º Semestre). A altura desta subarvore é 2.



Os principais métodos de uma árvore são:

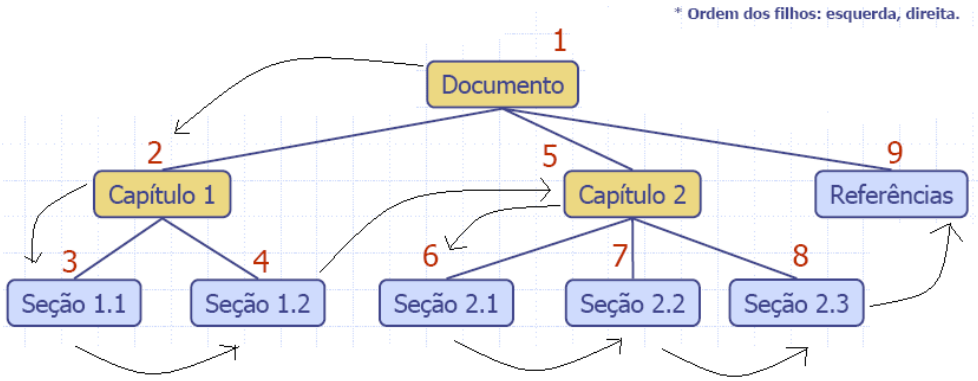
- **Raiz:** Retorna a raizda árvore
- **Pai(nodo):** Retorna o pai de um nodo. Ocorre um erro se nodo for a raiz.
- **Filho(nodo):** Retorna os filhos de um nodo.
- **Nodo_eh_Interno(nodo):** Testa se um nodo é do tipo interno.
- **Nodo_eh_externo(nodo):** Testa se um nodo é do tipo externo.
- **Nodo_eh_raiz(nodo):** Testa se um nodo é a raiz.
- **Tamanho:** Retorna a quantidade de nodos de uma árvore.

PERCURSO (OU CAMINHAMENTO) EM ÁRVORES

O percurso de uma árvore é a maneira ordenada de percorrer todos os nodos da árvore (percorrer todos os seus nós, sem repetir nenhum e sem deixar de passar por nenhum). É utilizada, por exemplo, para consultar ou alterar as informações contidas nos nós.

Percurso prefixado

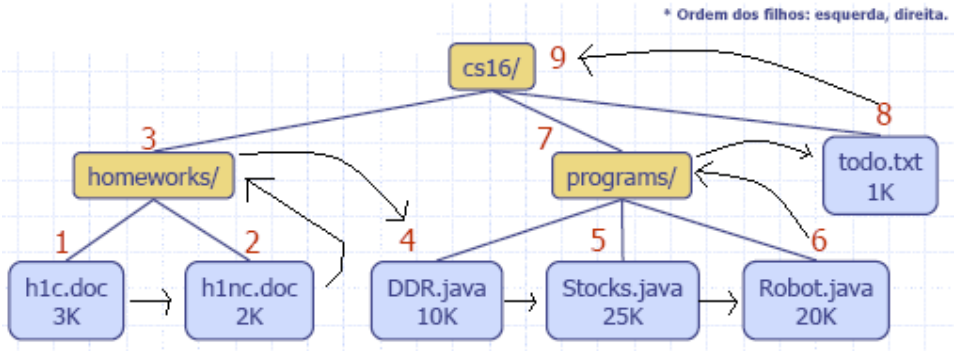
Um nodo é visitado antes de seus descendentes. Exemplo de aplicação: Imprimir um documento estruturado.



Os números em vermelho indicam a ordem em que os nodos são visitados. Caso fossem impressos, o resultado seria: Documento, Capítulo 1, Seção 1.1, Seção 1.2, Capítulo 2, Seção 2.1, Seção 2.2, Seção 2.3, Referências.

Percurso pós-fixado

Neste caminho, um nodo é visitado após seus descendentes. Exemplo de aplicação: Calcular o espaço ocupado por arquivos em pastas e subpastas.



Os números em vermelho indicam a ordem em que os nodos são visitados. Caso fossem impressos, o resultado seria: H1c.doc 3k, h1nc.doc 2k, homeworks/, DDR.java 10k, Stocks.java 25k, Robot.java 20k, programs/, todo.txt 1k, cs16/.

ÁRVORES BINÁRIAS

Material retirado da referência [2] e [3].

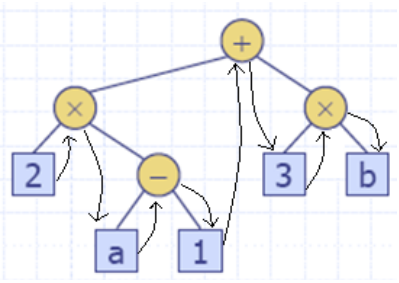
Uma **árvore binária** é uma árvore ordenada na qual todo nodo **tem, no máximo, dois filhos**. Uma árvore binária imprópria é aquela que possui apenas 1 filho. Já uma árvore binária própria é aquela em que todo nodo tem zero ou dois filhos, ou seja, todo nodo interno tem exatamente 2 filhos. Isso porque um nodo externo não tem filhos, ou seja, zero filhos. Para cada filho de um nodo interno, nomeamos cada filho como **filho da esquerda** e **filho da direita**. Esses filhos são ordenados de forma que o filho da esquerda venha antes do filho da direita.

A árvore binária suporta mais 3 métodos adicionais:

- **Filho_da_esquerda(nodo):** Retorna o filho da esquerda do nodo.
- **Filho_da_direita(nodo):** Retorna o filho da direita do nodo.
- **Irmão(nodo):** Retorna o irmão de um nodo

Caminhamento adicional para árvores binárias

Caminhamento interfixado: pode ser informalmente considerado como a visita aos nodos de uma árvore da esquerda para a direita. Para cada nodo **v**, o caminhamento interfixado visita **v** após todos os nodos da subárvore esqueda de **v** e antes de visitar todos os nodos da subárvore direita de **v**.

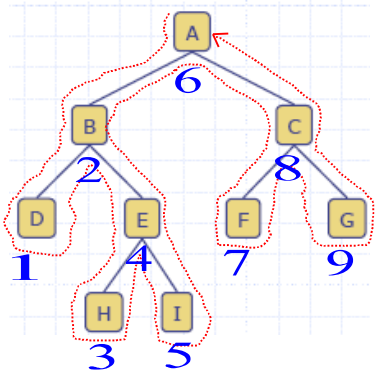


Os elementos acessados por este caminhamento formam a expressão:

2 x (a – 1) + (3 x b)

Os parênteses foram colocados para facilitar.

O Caminhamento de Euler: sobre uma árvore binária T pode ser informalmente definido como um "passeio" ao redor de T, no qual iniciamos pela raiz em direção ao filho da esquerda e consideramos as arestas de T como sendo "paredes" que devemos sempre manter nossa esquerda. Cada nodo de T é visitado três vezes pelo caminhamento de Euler.



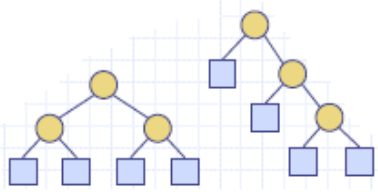
Prioridade das ações para efetuar o caminhamento:

- Ação "**pela esquerda**" (antes do caminho sobre a subárvore esquerda de v);
- Ação "**por baixo**" (entre o caminhamento sobre as duas subárvores de v);
- Ação "**pela direita**" (depois do caminhamento sobre a subárvore direita de v).

Propriedades de uma árvore binária

Seja **T** uma árvore binária (própria) com **n** nodos e seja **h** a altura de **T**. Então **T** tem as seguintes propriedades:

1. O número de nodos externos de T é pelo menos h+1 e no máximo 2^h.
2. O número de nodos internos de T é pelo menos h e no máximo $2^h - 1$.
3. O número total de nodos de T é pelo menos 2h +1 e no máximo $2^{h+1} - 1$.
4. A profundidade de T é pelo menos log(n+1) -1 e no máximo (n-1)/2.



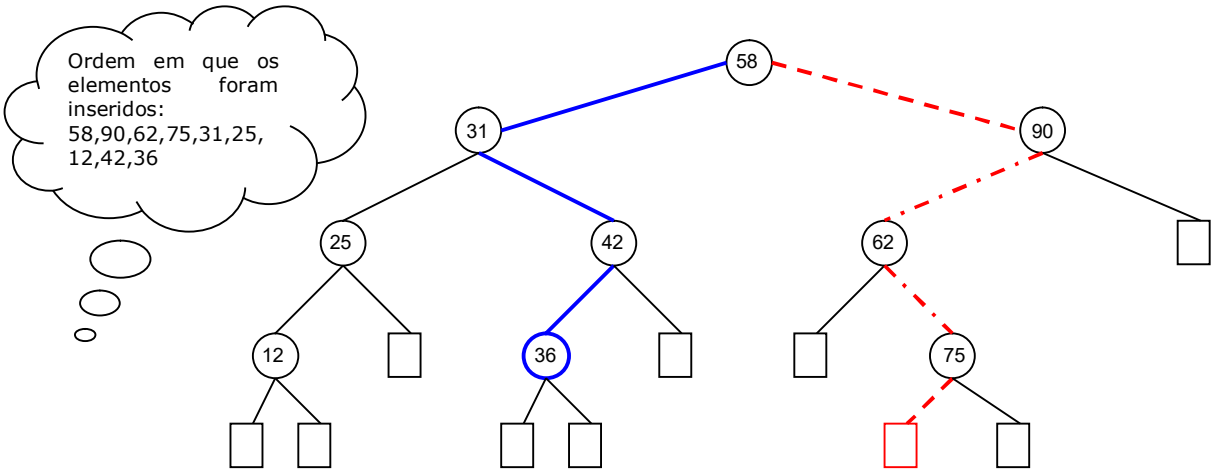
ÁRVORES BINÁRIAS DE BUSCA

Material retirado da referência [2] e [3].

Uma árvore de pesquisa binária é uma árvore binária em que **todo nó interno contém um registro**, e, para cada nó, todos os registros com chaves menores estão na subárvore esquerda e todos os registros com chaves maiores estão na subárvore direita.

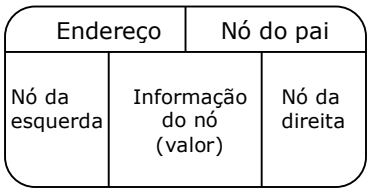
Podemos usar uma árvore binária de pesquisa **T** para localizar um elemento com um certo valor **x** percorrendo para baixo a árvore **T**. Em cada nodo interno, comparamos o valor do nodo corrente com o valor do elemento **x** sendo pesquisado.

- Se a resposta da questão for "é menor", então a pesquisa continua na subárvore esquerda.
- Se a resposta for "é igual", então a pesquisa terminou com sucesso.
- Se a resposta for "é maior", então a pesquisa continua na subárvore direita.
- Se encontrarmos um nodo externo (que é vazio), então a pesquisa terminou sem sucesso.



A figura acima representa uma árvore binária de pesquisa que armazena inteiros. O caminho indicado pela linha azul corresponde ao caminhamento ao procurar (com sucesso) 36. A linha pontilhada vermelha corresponde ao caminhamento ao procurar (sem sucesso) por 70. **Observe que o tempo de execução da pesquisa em uma árvore binária de pesquisa T é proporcional à altura de T.**

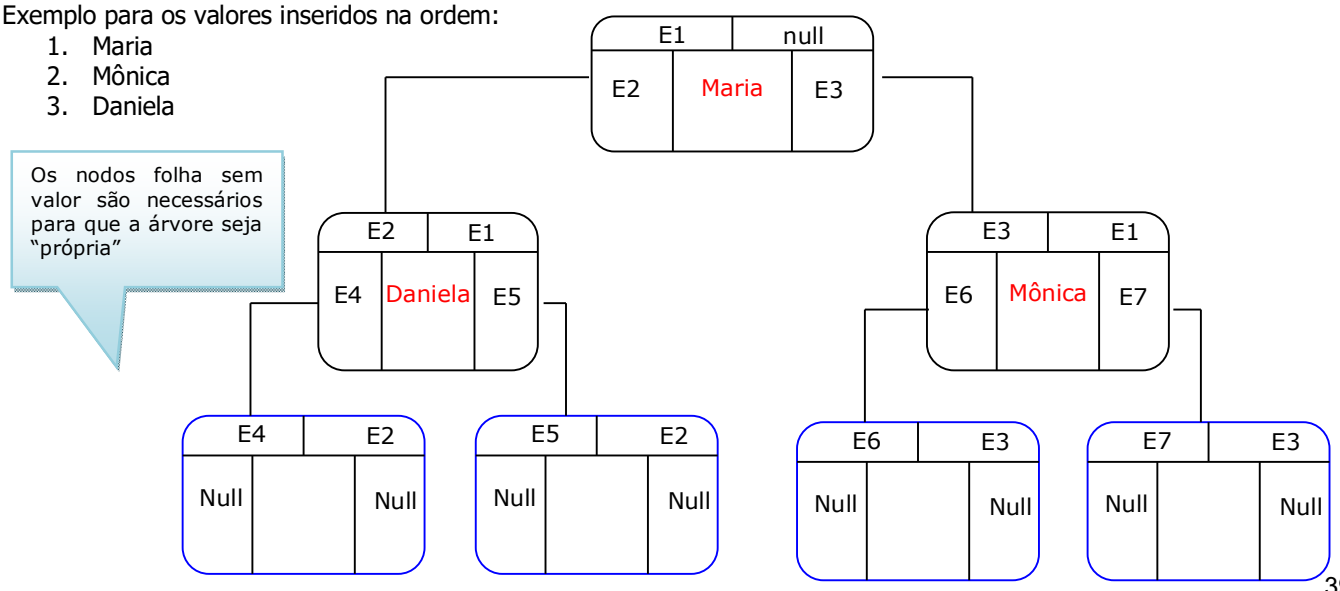
Estrutura para armazenar um nodo da árvore binária:



Exemplo para os valores inseridos na ordem:

1. Maria
2. Mônica
3. Daniela

Os nodos folha sem valor são necessários para que a árvore seja "própria"



Algoritmo para pesquisar um valor em uma árvore binária de pesquisa:

```
Pesquisa(  nodo, valor_pesquisado ) : Retorno
Inicio
        se Nodo_eh_externo(Nodo) = verdadeiro então
                escreva( 'Erro: Valor procurado não está na árvore!');
                pesquisa := null
        caso contrário
                Se valor_pesquisado < nodo.valor então
                        pesquisa ( nodo.esquerda, valor_pesquisado )
                caso contrário se valor_pesquisado > nodo.valor então
                        pesquisa ( nodo.direita, valor_pesquisado )
                caso contrário
                        pesquisa := nodo.valor;
Fim
```

Algoritmo para inserir um valor em uma árvore binária de pesquisa:

```
Insere( nodo, NovoValor )
Início
        se Nodo_eh_externo(nodo) = verdadeiro então
                CriaNodoExterno( nodo.esquerda )
                CriaNodoExterno( nodo.direita )
                Nodo.valor := NovoValor
        caso contrário
                se NovoValor  < nodo.valor então
                    Insere ( nodo.esquerda , NovoValor)
                caso contrário se NovoValor > nodo.valor então
                    Insere ( nodo.direita , NovoValor)
                caso contrário
                    escreva("O valor já existe na árvore.");
Fim;
```

O método CriaNodoExterno cria um nodo externo (sem valor e sem filhos)

Remoção de um nodo da árvore binária de pesquisa

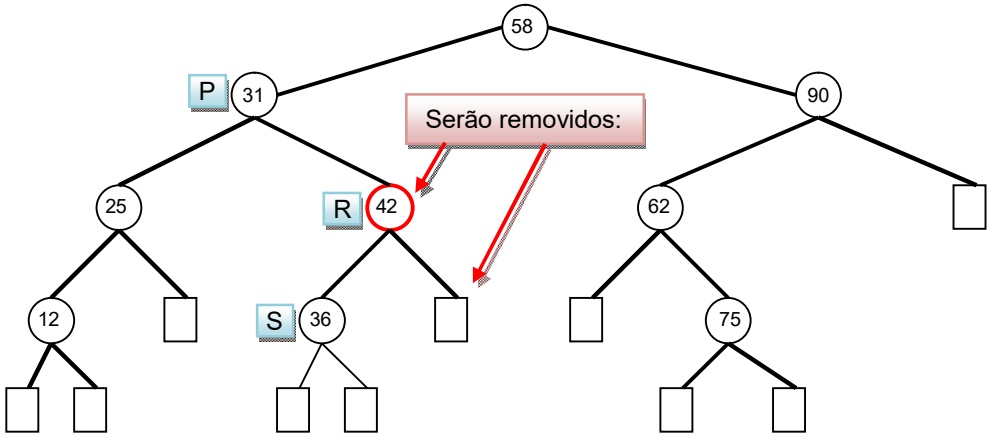
Ao removermos um elemento da árvore, precisamos manter a ordem. Abaixo iremos apresentar 2 situações de remoção:

Legenda:
R = nodo que se deseja remover
P = Nodo pai de R
S = nodo que irá substituir o nodo Excluído na árvore

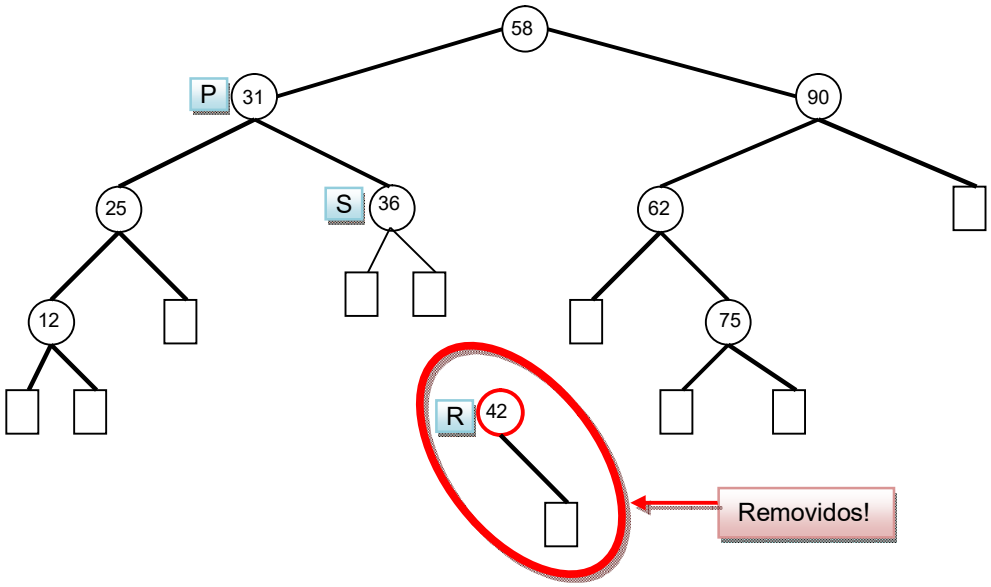
1. **Se pelo menos um dos filhos do nodo a ser removido (R) for um nodo externo:**

   Vamos remover o nodo com valor **42**.

- Identificamos o nodo a ser removido (R), o nodo substituto (S) e o novo pai do nodo a ser removido (P):



- Alteramos o Pai do nodo que será o substituto (S) do nodo removido (R).
  Também precisamos corrigir a referência no filho esquerdo ou direito do nodo pai (P) do nodo apagado (R).
  Caso o nodo excluído seja a raiz, precisamos atualizar a variável que faz referência a ela!

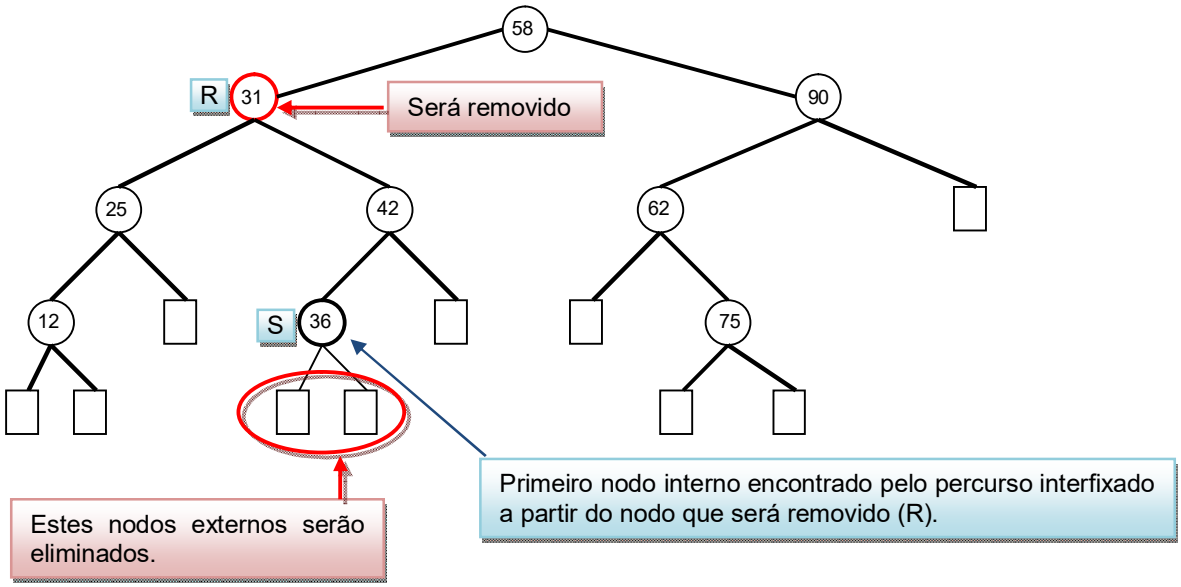


Legenda:
R = nodo que se deseja remover
S = nodo que irá substituir o nodo Excluído na árvore

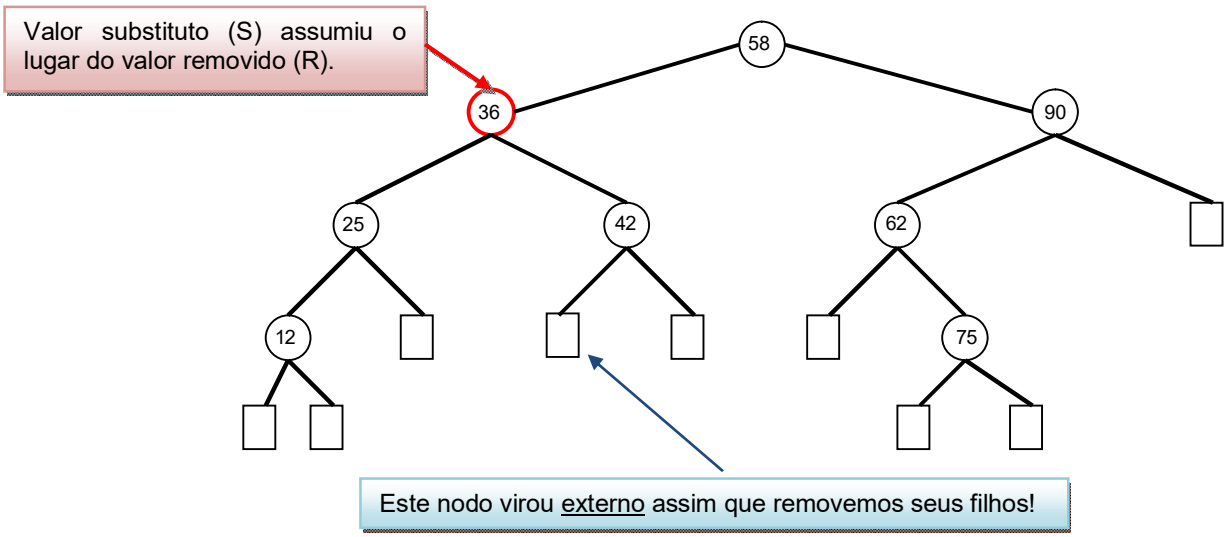
2. **Se os dois filhos do nodo a ser removido (R) forem nodos internos:**
Neste caso, não podemos simplesmente remover o nodo R da árvore, uma vez que isso criaria um "buraco" nela. Ao invés disso, faremos como segue abaixo:

Como exemplo, iremos remover o nodo 31:

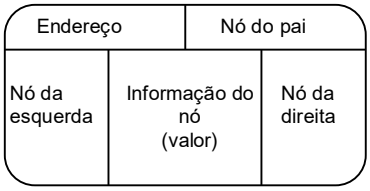
- Encontramos o primeiro nodo interno que segue o nodo a ser removido (R) utilizando o percurso interfixado. Este nodo será o nodo substituto (S)



- O valor de do elemento substituto deverá ser atribuído ao valor do elemento que foi removido.



Implementação de uma Árvore Binária de Busca em C#




```
// classe para reprentar 1 Nodo na árvore
class Nodo
{
    private Nodo no_pai = null;
    private Nodo no_direita = null;
    private Nodo no_esquerda = null;
    private int valor = 0;

    public int GetValor() { return valor; }

    public void SetValor(int v) { valor = v; }

    public void SetNoPai(Nodo no) { no_pai = no; }

    public void SetNoDireita(Nodo no) { no_direita = no; }

    public void SetNoEsquerda(Nodo no) { no_esquerda = no; }

    public Nodo GetNoPai() { return no_pai; }

    public Nodo GetNoDireita() { return no_direita; }

    public Nodo GetNoEsquerda() { return no_esquerda; }

    public Boolean NoEhRaiz(Nodo no) { return no.GetNoPai() == null; }

    /// <summary>
    /// Verifica se o nodo é externo
    /// </summary>
    /// <param name="no"></param>
    /// <returns></returns>
    public bool EhExterno()
    {
        return (no_direita == null) && (no_esquerda == null);
    }

    /// <summary>
    /// Verifica se o nodo é interno
    /// </summary>
    /// <param name="no"></param>
    /// <returns></returns>
    public bool EhInterno()
    {
        return (no_direita != null) || (no_esquerda != null);
    }


    /// <summary>
    /// Cria um nodo externo.
    /// </summary>
    /// <param name="Nopai"></param>
    /// <returns></returns>
    public static Nodo CriaNoExterno(Nodo Nopai)
    {
        Nodo no = new Nodo();
        no.SetNoPai(Nopai);
        return no;
    }
}
```

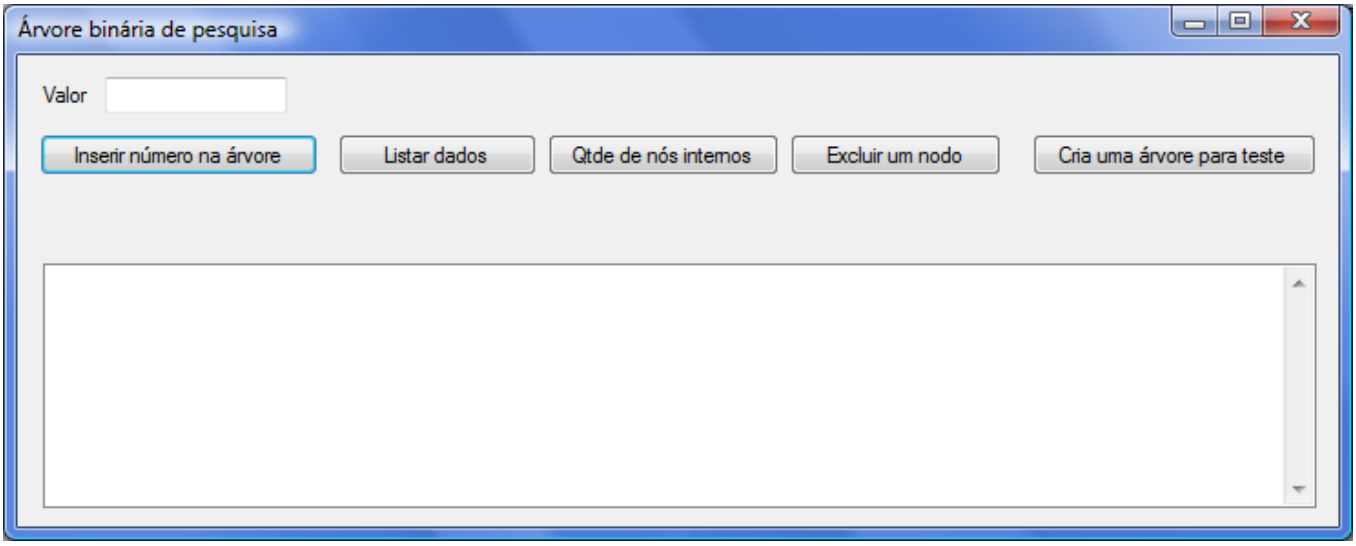
```
class ArvoreBin
{
    private Nodo raiz = null; // raiz da árvore
    private int qtdeNodosInternos = 0; // qtde de nos internos
    private string resultado = ""; // utilizada na listagem dos nodos
```

```
        /// <summary>
        /// Returna a qtde de nós internos
        /// </summary>
        /// <returns></returns>
        public int QtdeNodosInternos() // devolve a qtde de nós internos
        {
            return qtdeNodosInternos;
        }


        /// <summary>
        /// Insere um valor na árvore. Não aceita valores repetidos!!!
        /// </summary>
        /// <param name="valor">valor a ser inserido</param>
        public void Insere(int valor) // insere um valor int
        {
            Nodo no_aux;

            if (qtdeNodosInternos == 0) // árvore vazia!
            {
                // árvore vazia, devemos criar o primeiro Nodo, que será a raiz
                no_aux = new Nodo();
                raiz = no_aux;
            }
            else
            {
                // localiza onde deve ser inserido o novo nó.
                no_aux = PesquisaValor(valor, raiz);
                if (no_aux.EhInterno())
                {
                    throw new Exception("Este valor já existe na árvore!!!!");
                }
            }
            // este era um Nodo externo e portanto não tinha filhos.
            // Agora ele passará a ser interno, portanto devemos criar outros 2
            // nodos externos (filhos) para ele.
            no_aux.SetValor(valor);
            no_aux.SetNoEsquerda(Nodo.CriaNoExterno(no_aux));
            no_aux.SetNoDireita(Nodo.CriaNoExterno(no_aux));
            qtdeNodosInternos++;
        }



        private void PercursoInterfixado(Nodo no)
        {
            if (no.EhExterno())
                return;

            PercursoInterfixado (no.GetNoEsquerda());
            resultado = resultado + " - " + Convert.ToInt32(no.GetValor());
            PercursoInterfixado (no.GetNoDireita());
        }


        /// <summary>
        /// Devolve um string com os elementos da árvore, em ordem crescente
        /// </summary>
        /// <returns></returns>
        public string ListagemEmOrdem()
        {
            resultado = "";
            if (qtdeNodosInternos != 0)
                PercursoInterfixado(raiz);
            return resultado;
        }



        /// <summary>
        /// Pesquisa um nodo na árvore e devolve o nodo. Caso não encontre, devolve o nodo
        /// externo onde a pesquisa parou.
        /// </summary>
        /// <param name="valor"></param>
        /// <param name="no"></param>
        /// <returns></returns>
        private Nodo PesquisaValor(int valor, Nodo no)
        {
            if (no.EhExterno())
                return no; // não achou!
            else if (no.GetValor() == valor)
```

```
            return no;
        else if (valor > no.GetValor())
            return PesquisaValor(valor, no.GetNoDireita());
        else
            return PesquisaValor(valor, no.GetNoEsquerda());
    }



    /// <summary>
    /// Remove um valor da árvore
    /// </summary>
    /// <param name="valor"></param>
    public void Remove(int valor)
    {
        //primeiro, procuramos o nodo que tem o valor:
        Nodo noQueSeraApagado = PesquisaValor(valor, raiz);

        if (noQueSeraApagado == null || noQueSeraApagado.EhExterno())
            throw new Exception("Valor não existe na árvore");
        else
        {
            // um dos filhos é um nó externo
            if (noQueSeraApagado.GetNoEsquerda().EhExterno() ||
                noQueSeraApagado.GetNoDireita().EhExterno())
            {
                ExcluiComNodoExterno(noQueSeraApagado);
            }
            else
            {
                ExcluiSemNodoExterno(noQueSeraApagado);
            }
        }
    }


    /// <summary>
    /// Exclui um nodo que abaixo dele possua, ao menos, 1 nodo exteno.
    /// </summary>
    /// <param name="noQueSeraApagado"></param>
    private void ExcluiComNodoExterno(Nodo noQueSeraApagado)
    {
        qtdeNodosInternos--;
        //descobre quem será o nodo que irá ficar no lugar do que foi apagado
        Nodo nodo_substituto;
        if (noQueSeraApagado.GetNoEsquerda().EhInterno())
            nodo_substituto = noQueSeraApagado.GetNoEsquerda();
        else
            nodo_substituto = noQueSeraApagado.GetNoDireita();

        // substitui o apagado pelo novo nodo
        Nodo PaiNodoApagado = noQueSeraApagado.GetNoPai();

        //altera o pai do novo substituto
        nodo_substituto.SetNoPai(PaiNodoApagado);

        //o pai do nodo substituto também deve ter sua referência de filho corrigida.
        //mas primeiro precisamos saber o no apagado er a raiz, que neste caso não tem pai.
        if (PaiNodoApagado != null)
        {
            if (PaiNodoApagado.GetNoDireita() == noQueSeraApagado)
                PaiNodoApagado.SetNoDireita(nodo_substituto);
            else
                PaiNodoApagado.SetNoEsquerda(nodo_substituto);
        }
        else
            raiz = nodo_substituto;
    }



    /// <summary>
    /// Pesquisa o próximo nodo Interno seguindo o percurso interfixado.
    /// </summary>
    /// <param name="no"></param>
    /// <returns></returns>
    private Nodo PesquisaNodoInternoInterfixado(Nodo no)
    {
        if (no.EhExterno())
            return null;

        Nodo retorno = PesquisaNodoInternoInterfixado(no.GetNoEsquerda());
        if (retorno == null)
```

```
                return no;
            else
                return retorno;
        }

        /// <summary>
        /// Exclui um nodo que abaixo dele não há nodos externos.
        /// </summary>
        /// <param name="noQueSeraApagado"></param>
        private void ExcluiSemNodoExterno(Nodo noQueSeraApagado)
        {
            //encontra o nodo substituto
            Nodo NodoSubstituto = PesquisaNodoInternoInterfixado(noQueSeraApagado.GetNoDireita());
            Console.Write("Nodo substituto: " + NodoSubstituto.GetValor());

            //Altera o valor do nodo que será removido pelo valor do nodo substituto
            noQueSeraApagado.SetValor(NodoSubstituto.GetValor());

            //Remove o nodo substituto
            ExcluiComNodoExterno(NodoSubstituto);
        }
    }
```

Interface com o Usuário:



Código da interface com o usuário:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace ArvoreBinaria_visual
{

    public partial class Form1 : Form
    {
        private ArvoreBin minhaArvore = new ArvoreBin();

        public Form1()
        {
            InitializeComponent();
        }

        private void btnInsere_Click(object sender, EventArgs e)
        {
            try
            {
```

```
                minhaArvore.Insere(Convert.ToInt32(txtValor.Text));
                listBox1.Items.Add("Inserido: " + txtValor.Text);
            }
            catch (Exception erro)
            {
                MessageBox.Show(erro.Message);
            }
            txtValor.Clear();
            txtValor.Focus();
        }

        private void btnListar_Click(object sender, EventArgs e)
        {
            string r = minhaArvore.ListagemEmOrdem();
            if (r.Length == 0)
                listBox1.Items.Add("Árvore vazia!");
            else
                listBox1.Items.Add(r);
        }

        private void btnQtdeNosInternos_Click(object sender, EventArgs e)
        {
            listBox1.Items.Add("Qtde: " + minhaArvore.QtdeNodosInternos());
        }


        private void btnRemover_Click(object sender, EventArgs e)
        {
            try
            {
                minhaArvore.Remove(Convert.ToInt16(txtValor.Text));
            }
            catch (Exception erro)
            {
                MessageBox.Show(erro.Message);
            }
        }

        private void btnArvoreTeste_Click(object sender, EventArgs e)
        {   //Valores para criar uma árvore com o objetivo de facilitar os testes...
            minhaArvore.Insere(58);
            minhaArvore.Insere(31);
            minhaArvore.Insere(25);
            minhaArvore.Insere(42);
            minhaArvore.Insere(12);
            minhaArvore.Insere(36);
            minhaArvore.Insere(90);
            minhaArvore.Insere(62);
            minhaArvore.Insere(75);
        }
    }
}
```

Exercícios:

1-) Altere o programa principal e a árvore para atender as seguintes solicitações:

a- Pesquise um valor.
b- Calcule a altura da árvore.
c- Crie uma variável para guardar a qtde de nodos externos.
d- Faça o percurso pré-fixado (pré-ordem)
e- Faça o percurso pós-fixado (pós-ordem)

2-) Crie uma nova árvore onde seja possível armazenar objetos da classe funcionário (int codigo, string nome) e faça o programa principal de forma que seja possível adicionar objetos da classe funcionário nessa árvore. Altere o método de efetuar pesquisa interfixada para que ele devolva uma lista (a lista criada por você, não a lista do C# da classe List) com os funcionários armazenados na árvore. Liste os funcionários no form principal. Ao adicionar na árvore, utilize como chave o nome do funcionário. Não permite nomes repetidos.

3-) Tire uma cópia do exercício 2, alterando a classe nodo de forma que ela armazenar objetos da classe Object. O objetivo é poder armazenar qualquer tipo de objeto. Então, faça as alterações necessárias para que ela continue funcionando com os objetos da classe Funcionario.

ORDENAÇÃO

Material retirado de:
Referência [2], [3]
http://pt.wikipedia.org/wiki/Bubble_sort (com exemplos em várias linguagens)

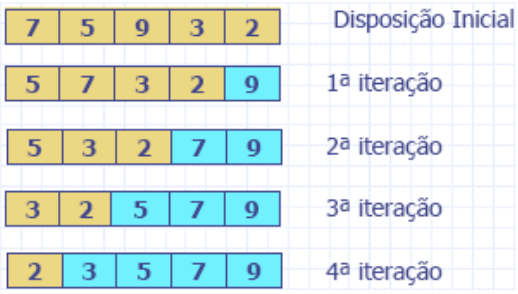
Ordenar corresponde ao processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente. O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado. Imagine como seria difícil utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética!
Existem diversos métodos para realizar ordenação. Iremos estudar aqui dois dos principais métodos.

Bubble sort

O bubble sort, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o menor elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.
No melhor caso, o algoritmo executa N operações relevantes, onde n representa o número de elementos do vetor. No pior caso, são feitas n^2 operações. A complexidade desse algoritmo é de Ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

O algoritmo pode ser descrito em pseudo-código como segue abaixo. V é um VETOR de elementos que podem ser comparados e n é o tamanho desse vetor.

```
BUBBLESORT (V[], n)
1    houveTroca := verdade  # uma variável de controle
2    enquanto houveTroca for verdade faça
3         houveTroca := falso
4         para i de 1 até n-1 faça
5              se V[i] é maior que  V[i + 1]
6                   então troque V[i] e V[i + 1] de lugar e
7                         houveTroca := verdade
```



Implementação em C# utilizando For e While

```
class C_BubbleSort
    {
        static int[] Ordena_BubbleSort(int[] vetor)
        {
            int aux;

            for (int i = vetor.Length - 1; i >= 1; i--)
            {
                for (int j = 0; j <= i - 1; j++)
                {
                    if (vetor[j] > vetor[j + 1])
                    {
                        //efetua a troca de valores
                        aux = vetor[j];
                        vetor[j] = vetor[j + 1];
                        vetor[j + 1] = aux;
                    }
                }
            }
            return vetor;
        }

        static void Main(string[] args)
        {
            int[] dados = new int[10];

            for (int i = 0; i < dados.Length; i++)
            {
                Console.WriteLine("Informe um número");
                dados[i]=Convert.ToInt16(Console.ReadLine());
            }

            Ordena_BubbleSort(dados);
```

```
class C_BubbleSort
{
    static int[] Ordena_BubbleSort(int[] vetor)
    {
        int aux;
        bool houvetroca;

        do
        {
            houvetroca = false;
            for (int j = 0; j <= vetor.Length - 2; j++)
            {
                if (vetor[j] > vetor[j + 1])
                {
                    //efetua a troca de valores
                    houvetroca = true;
                    aux = vetor[j];
                    vetor[j] = vetor[j + 1];
                    vetor[j + 1] = aux;
                }
            }
        }
        while (houvetroca == true);

        return vetor;
    }

    static void Main(string[] args)
    {
        int[] dados = new int[10];

        for (int i = 0; i < dados.Length; i++)
        {
            Console.WriteLine("Informe um número");
```

```
            Console.WriteLine("\n\nDados ordenados:");
            for (int i = 0; i < dados.Length; i++)
            {
                Console.WriteLine( dados[i] );
            }
            Console.ReadKey();
        }
    }
```

```
            dados[i]= Convert.ToInt16(Console.ReadLine());
        }

        Ordena_BubbleSort(dados);

        Console.WriteLine("\n\nDados ordenados:");
        for (int i = 0; i < dados.Length; i++)
        {
            Console.WriteLine(dados[i]);
        }

        Console.ReadKey();
    }
}
```

Quicksort

O algoritmo **Quicksort** é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960, quando visitou a Universidade de Moscou como estudante. Foi publicado em 1962 após uma série de refinamentos.

O Quicksort adota a estratégia de divisão e conquista. Os passos são:

1. Escolha um elemento da lista, denominado *pivô (de forma randômica)*;
2. Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores ou iguais a ele, e todos os elementos posteriores ao pivô sejam maiores ou iguais a ele. Ao fim do processo o pivô estará em sua posição final. Essa operação é denominada *partição*;
3. Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;

A base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

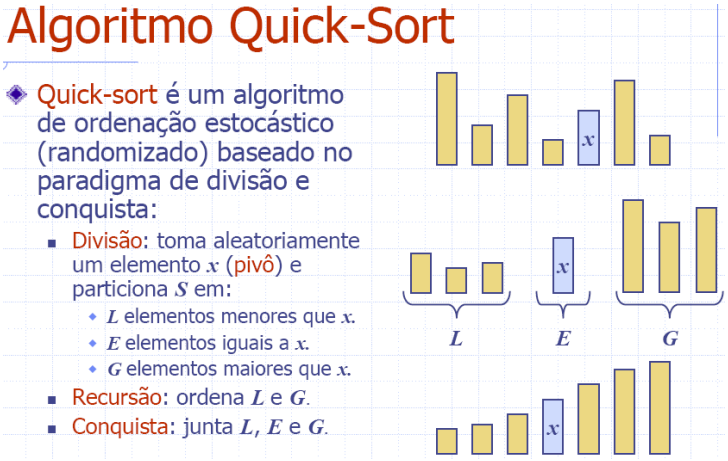
Complexidade

- Ω (n Log_2 n) no melhor caso e no caso médio θ
- $O(n^2)$ no pior caso;

Implementações

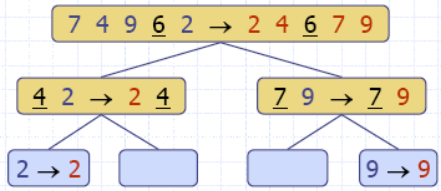
Algoritmo em português estruturado

```
proc quicksort (x:vet[n] int; ini:int; fim:int; n:int)
var
   int: i,j,y,aux;

início
   i <- ini;
   j <- fim;
   y <- x[(ini + fim) div 2];
   repete
       enquanto (x[i] < y) faça
          i <- i + 1;
       fim-enquanto;
       enquanto (x[j] > y) faça
          j <- j - 1;
       fim-enquanto;
       se (i <= j) então
          aux <- x[i];
          x[i] <- x[j];
          x[j] <- aux;
          i <- i + 1;
          j <- j - 1;
       fim-se;
    até_que (i >= j);
    se (j > ini) então
        exec quicksort (x, ini, j, n);
    fim-se;
    se (i < fim) então
        exec quicksort (x, i, fim, n);
    fim-se;
fim.
```



Árvore Quick-Sort

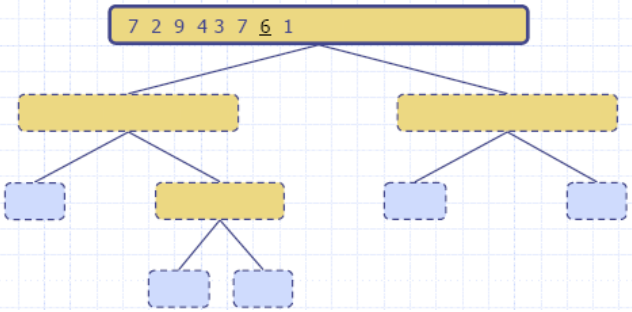
- Uma execução do quick-sort pode ser representada através de uma árvore binária:
  - A raiz representa a chamada inicial.
  - Os demais nós representam chamadas recursivas do algoritmo.
  - As folhas representam chamadas para subseqüências de tamanho 0 ou 1.



33

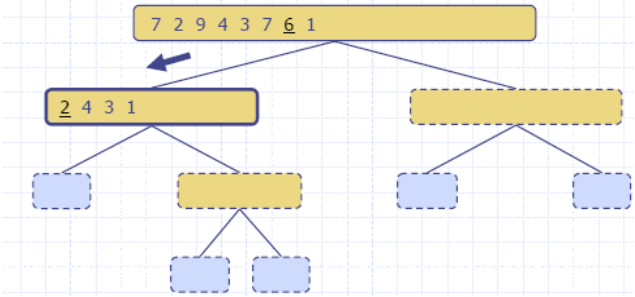
Exemplo de Execução

- Seleção do pivô:



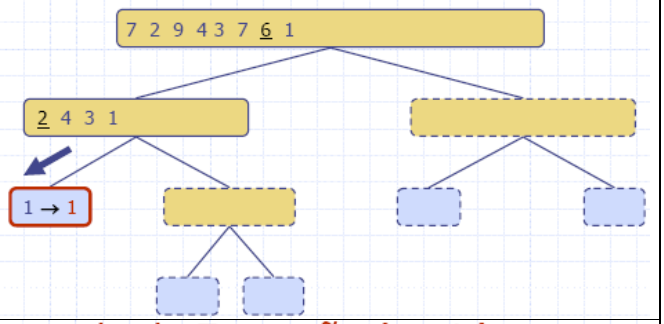
Exemplo de Execução (cont.)

- Partição, chamada recursiva e seleção do pivô:



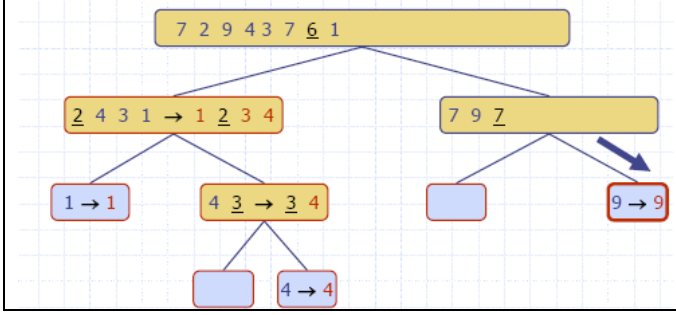
Exemplo de Execução (cont.)

- Partição, chamada recursiva e caso básico:



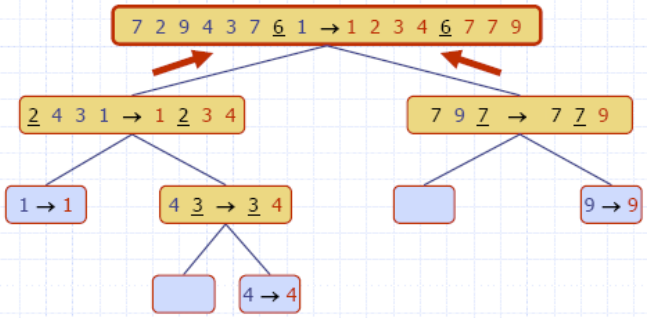
Exemplo de Execução (cont.)

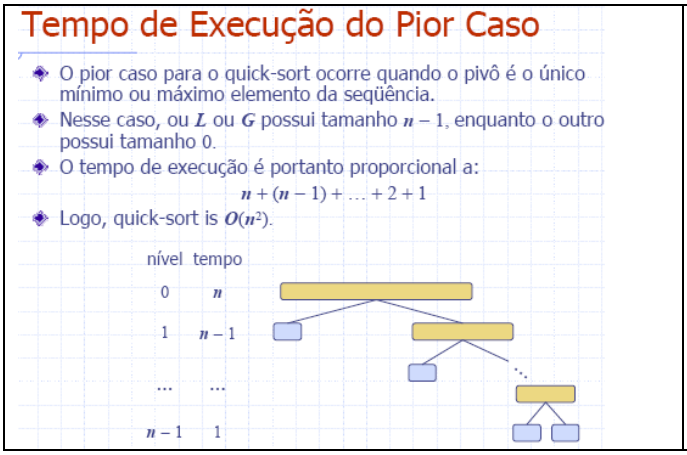
- Partição, ..., chamada recursiva, caso básico:

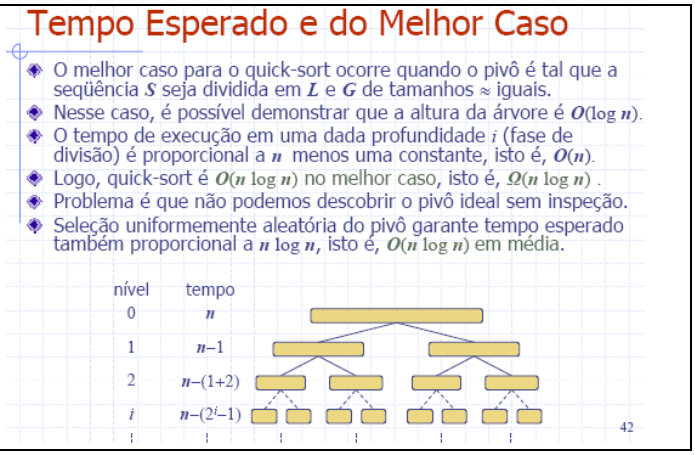


Exemplo de Execução (cont.)

- Junção, junção







Implementação em C#

Método que efetua a ordenação

```
static void QuickSort(int[] vetor, int esq, int dir)
{
    int pivo, aux, i, j;
    int meio;

    i = esq;
    j = dir;

    meio = (int)((i + j) / 2);
    pivo = vetor[meio];

    do
    {
        while (vetor[i] < pivo) i = i + 1;
        while (vetor[j] > pivo) j = j - 1;

        if (i <= j)
        {
            aux = vetor[i];
            vetor[i] = vetor[j];
            vetor[j] = aux;
            i = i + 1;
            j = j - 1;
        }
    }
    while (j > i);

    if (esq < j) QuickSort(vetor, esq, j);
    if (i < dir) QuickSort(vetor, i, dir);
}
```

Método MAIN que solicita os números e chama o método quicksort para ordenar.

```
static void Main(string[] args)
{
    int[] dados = new int[10];

    Console.WriteLine("Entre com {0} números", dados.Length);
    for (int i = 0; i < dados.Length; i++)
        dados[i] = Convert.ToInt16(Console.ReadLine());

    QuickSort(dados, 0, dados.Length - 1);
```

```
            Console.WriteLine("\n\nNúmeros ordenados:\n");
            for (int i = 0; i < dados.Length; i++)
                Console.WriteLine ( dados[i]);

            Console.ReadLine();
        }
```

Exercícios:

1 - Faça o quicksort dos numeros abaixo. Para pivo, escolha sempre o primeiro numero.

a) 25.31.42.33.98.21.3.16.58.7

b) 1.3.4.5.10.15.16.17

PESQUISA EM MEMÓRIA PRIMÁRIA

Pesquisa sequencial

Retirado de : http://pucrs.campus2.br/~annes/alg3_pesqseq.html
[2]

O método de pesquisa mais simples que existe funciona da seguinte forma: a partir do primeiro registro, pesquise sequencialmente até encontrar a chave procurada; então pare. A complexidade desta pesquisa no pior caso é *n*, onde *n* é o tamanho total do vetor sendo pesquisado.

```
{Algoritmo em Pascal}
Function PesquisaSequencial(vetor : array of Integer, chave,n : integer) :  integer;
Var i: integer;
    Achou : boolean;
Begin
        PesquisaSequencial := -1; { significa que não encontrou }
        Achou := false;
        i:= 1;
        Repeat
                If vetor[i] = chave then
                Begin
                        Achou := true;
                        PesquisaSequencial := i;
                End;
                i := i + 1;
        Until (i > n) or (achou = true);
End;
```

Dado o exemplo:

5	7	1	9	3	21	15	99	4	8

No exemplo acima, seriam necessárias 7 iterações para encontrar o valor 15.

Pesquisa Binária

Retirado de : http://pt.wikipedia.org/wiki/Pesquisa_bin%C3%A1ria
[2]

A pesquisa ou busca binária (em inglês binary search algorithm ou binary chop) é um algoritmo de busca em vetores que requer acesso aleatório aos elementos do mesmo. **Ela parte do pressuposto de que o vetor está ordenado**, e realiza sucessivas divisões do espaço de busca comparando o elemento buscado (chave) com o elemento no meio do vetor. Se o elemento do meio do vetor for a chave, a busca termina com sucesso. Caso contrário, se o elemento do meio vier antes do elemento buscado, então a busca continua na metade posterior do vetor. E finalmente, se o elemento do meio vier depois da chave, a busca continua na metade anterior do vetor. A complexidade desse algoritmo é da ordem de $O(\log_2 n)$, onde n é o tamanho do vetor de busca.

Um pseudo-código recursivo para esse algoritmo, dados V o vetor com elementos comparáveis, n seu tamanho e e o elemento que se deseja encontrar:

```
BUSCA-BINÁRIA (V[], inicio, fim, e}
     i recebe o índice no meio de inicio e fim
     se V[i] é igual a e
          então devolva o índice i     # encontrei e
          senão se V[i] vem antes de e
                    então faça a BUSCA-BINÁRIA(V, i+1, fim, e)
                    senão faça a BUSCA-BINÁRIA(V, inicio, i-1, e)
```

{Algorítmo em Pascal}

```
function BuscaBinaria (Vetor: array of string; Chave: string; Dim: integer): integer;
     var inicio, fim: integer; {Auxiliares que representam o inicio e o fim do vetor analisado}
         meio: integer; {Meio do vetor}
     begin
          fim := Dim; {O valor do último índice do vetor}
          inicio := 1; {O valor do primeiro índice do vetor}
            repeat
               meio := (inicio+fim) div 2;
                 if (Chave = vetor[meio]) then
                      BuscaBinaria := meio;
                 if (Chave < vetor[meio]) then
                      fim:=(meio-1);
                 if (Chave > vetor[meio]) then
                      inicio:=(meio+1);
          until (Chave = Vetor[meio]) or (inicio > fim);
          if (Chave = Vetor[meio]) then
                BuscaBinaria := meio
          else
                BuscaBinaria := -1; {Retorna o valor encontrado, ou -1 se a chave nao foi encontrada.}
     end;
```

Exemplo: Dado vetor abaixo, **Procurar o número 80:**

1	3	7	9	**15**	17	21	65	80	99

Meio = 15
80 é maior que 15, então, procurar no intervalo à direita:

17	21	**65**	80	99

Meio = 65
80 é maior que 65, então procurar no intervalo à direita:

80	99

Meio = 80
É o valor procurado. Parar a pesquisa!

80

Para procurar 80, foram necessárias 3 iterações.

Árvores de pesquisa

Vide Árvores Binárias de Busca.

Exercícios:

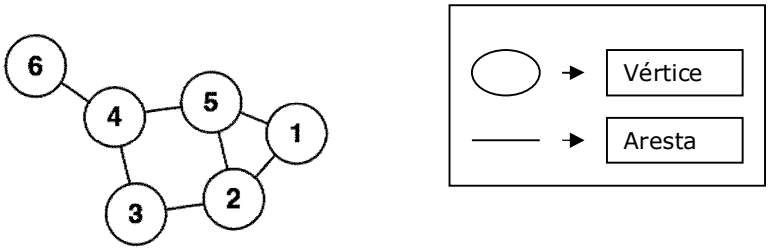
1) Implemente em C# a pesquisa binária. Faça um programa (windows forms /console) para testar a pesquisa.

GRAFOS

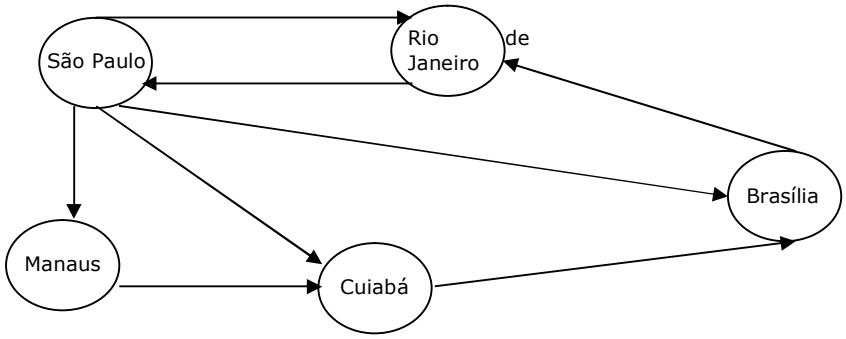
Material sobre grafos: [3], [4], [5]

http://www.inf.ufsc.br/grafos/livro.html
http://www.ime.usp.br/~pf/algoritmos_para_grafos/

Um grafo é um conjunto de pontos, chamados vértices (ou nodos ou nós), conectados por linhas, chamadas de arestas (ou arcos). Dependendo da aplicação, arestas podem ou não ter direção, pode ser permitido ou não arestas ligarem um vértice a ele próprio e vértices e/ou arestas podem ter um peso (numérico) associado. Se todas as arestas têm uma direção associada (indicada por uma seta na representação gráfica) temos **um grafo dirigido**, **dígrafo** ou **grafo orientado.** Se todas as arestas em um grafo foram não-dirigidas, então dizemos que o grafo é um **grafo não-dirigido**. Um grafo que tem arestas não-dirigidas e dirigidas é chamado de **grafo misto**.



Exemplo de um grafo **não-dirigido** com 6 vértices e 7 arestas.



Exemplo de um grafo **dirigido** com 8 arestas e 5 vértices.

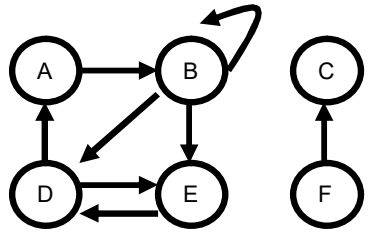
Algumas definições sobre grafos:

- **Grau**: número de arestas ligadas a um vértice.
- **Grau de entrada**: número de setas que chegam em um vértice X, in(X).
- **Grau de saída**: número de setas que saem de um vértice X, out(X).
- **Fonte**: todo vértice, cujo grau de entrada é 0 (zero).
- **Sumidouro (poço)**: todo vértice, cujo grau de saída é 0 (zero).

Sendo Grafo dirigido abaixo G uma dupla (N,A), em que N representa um conjunto finito de nós ou vértices que compõe G, A relação binária que pode existir entre os nós é chamada de arco ou aresta.

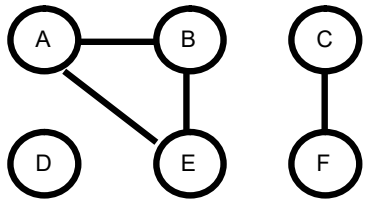
G = (N,A)

N = Conjunto de nós (ou vértices) de G
A = Conjunto de arcos (ou arestas) de A.



N = { A, B, C , D, E, F }
A = { (A,B), (B,B), (B,D), (B,E), (D,A), (D,E), (E,D), (F,C) }

Grafo **dirigido** - G1



N = { A, B, C , D, E, F }
A = { (A,B), (A,E), (B,E), (F,C) }

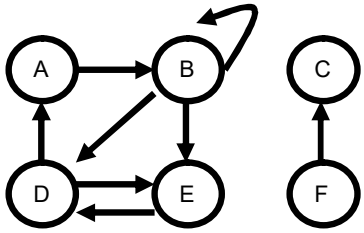
Grafo não-**dirigido - G2**

<u>Relação de incidência</u>

Entre arcos e nós pode ser definida uma <u>relação de incidência</u>. Neste caso, podemos dizer que um arco é <u>incidente do nó</u> x quando ele sai de x. Dizemos que um arco é <u>incidente para o nó</u> x quando ele chega a x.

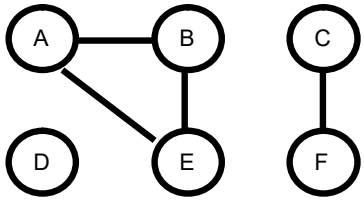
No caso do grafo G1 (dirigido)

- Arcos incidentes <u>do nó</u> "B" (saem de "B"): (B,B) (B,D) e (B,E)
- Arcos incidentes <u>para o nó</u> "B" (chegam em "B"): (A,B) e (B,B)



No caso do grafo G2 (não dirigido):

- Arcos incidentes no nó "B" : (A,B) e (B,E)
- Arcos incidentes no nó "C" : (C,F)
- Arcos incidentes no nó "F" : (C,F)

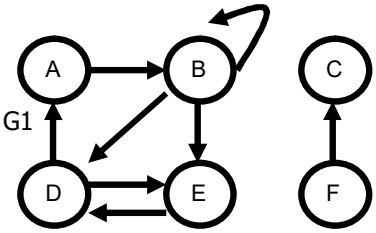


<u>Relação de adjacência</u>

Ocorre entre dois arcos se existir um arco interligando-os; em outras palavras, o arco deve pertencer ao conjunto A do grafo:

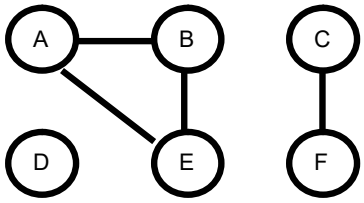
No grafo dirigido G1:

- O nó "B" é adjacente ao nó "A".
- O nó "A" **não** é adjacente ao "B" por que o arco (B,A) não pertence a G1
- O nó "E" é adjacente ao nó "B".



No grafo não dirigido G2 a, relação de adjacência é simétrica.

- O nó "B" é adjacente ao nó "A".
- O nó "A" é adjacente ao nó "B".



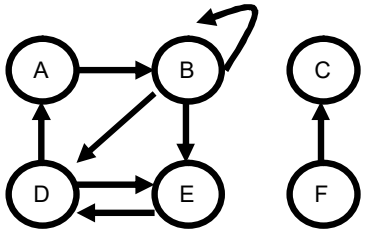
Matriz de adjacências para um grafo

Para elaborar a matriz de adjacências para um grafo G=(N,A), assume-se que os nós são numerados da seguinte forma: 1,2,3,..., N. Então, é construída uma matriz de adjacências MAdj para o grafo G=(N,A) com dimensões N x N e elementos $e_{i,j}$ cujos valores podem ser:

$$e_{i,j} = \begin{cases} 1, \text{ se } (i,j) \text{ pertence a A} \\ 0, \text{ caso contrário.} \end{cases}$$

Para o grafo dirigido abaixo, a matriz de adjacências seria a seguinte:

N = { A, B, C , D, E, F }
A = { (A,B), (B,B), (B,D), (B,E), (D,A), (D,E), (E,D), (F,C) }



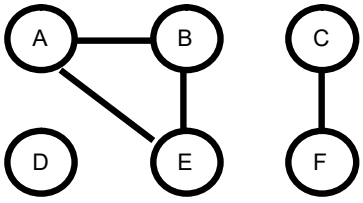
	A 0	B 1	C 2	D 3	E 4	F 5
A 0	0	**1**	0	0	0	0
B 1	0	**1**	0	**1**	**1**	0
C 2	0	0	0	0	0	0
D 3	**1**	0	0	0	**1**	0
E 4	0	0	0	**1**	0	0
F 5	0	0	**1**	0	0	0

A matriz de adjacência de um dígrafo tem colunas e linhas indexadas pelos vértices. Se adj é uma tal matriz então, para cada vértice v e cada vértice w,

adj[v][w] = 1 se v-w é um arco e
adj[v][w] = 0 em caso contrário.

As matrizes de adjacência de grafos não direcionados são simétricas: adj[v][w] = adj[w][v] para todo v e todo w. EX:

N = { A, B, C , D, E, F }
A = { (A,B), (A,E), (B,E), (F,C) }



	A 0	B 1	C 2	D 3	E 4	F 5
A 0	0	**1**	0	0	**1**	0
B 1	**1**	0	0	0	**1**	0
C 2	0	0	0	0	0	**1**
D 3	0	0	0	0	0	0
E 4	**1**	**1**	0	0	0	0
F 5	0	0	**1**	0	0	0

Percurso em um Grafo

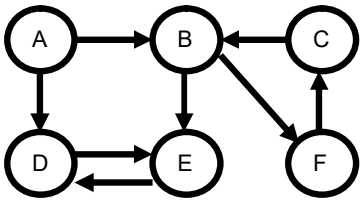
Percurso em um grafo é uma forma sistemática de realizar a exploração dos nós em um grafo com o objetivo de obter informação sobre sua estrutura. Devido à ausência de um ponto de referência tão claro como a raiz de uma árvore, o ponto de início do passeio é sempre visto como uma situação crítica, que deve ser bem tratada durante a elaboração de algoritmos de passeio sobre os grafos.

Também deve-se tomar o cuidado de não repetir a visita a um nó. As duas principais formas de passeio em grafos são a busca em largura e a busca em profundidade.

Busca em Largura ou Amplitude (*breadth-first search*)

Todos os nós são localizados a uma distância *d* de um nó *n*, escolhidos de forma aleatória, são percorridos antes dos nós localizados a uma distância *d+1* de *n*.

```
Algoritmo largura (no)
Inicio
        Visitar_no (no)
        Marcar_como_visitado(no)
        Fila.enfileira(no)
        Enquanto ( fila.vazia() == false) faça
                noAux = fila.Desenfileira()
                Para cada nó m adjacente a noAux faça
                        Se (m.no_visitado == false)
                                Visitar_no (m)
                                Marcar_como_visitado(m)
                                Fila.enfileira( m )
                        Fim se
                Fim para
        Fim Enquanto
Fim
```



A	→	B	D
B	→	E	F
C	→	B	
D	→	E	
E	→	D	
F	→	C	

Lista de Adiacência

Busca em Profundidade (*deepth-first search*)

Para um nó *n*, escolhido de forma aleatória, visita-se m de seus nós adjacentes. E para cada um desses nós que for visitado, visita-se um dos nós adjacentes, e assim por diante, até que o momento em que for encontrado um nó sem adjacentes. Então, ocorre um 'retorno' com o objetivo de visitar os nós restantes adjacentes a *n*, e o processo repete-se novamente. O algoritmo pode ser desenvolvido utilizando-se recursividade. Iremos apresentar abaixo a versão recursiva do algoritmo:

```
Algoritmo Profundidade (no)
Inicio
        Visitar_no (no)
        Marcar_como_visitado(no)
        Para cada nó m adjacente a no faça
                Se (m.no_visitado == false)
                        Profundidade(m)
                Fim se
        Fim para
        Fim Enquanto
Fim
```