Apostila de Estrutura de Dados e Algoritmos em C#

Prof. Ms. Eduardo R. Marcelino

ermarc@itelefonica.com.br

2009

Índice

Tipos Abstratos de Dados (TAD) 2
Tipos de Dados Concretos (TDC) 2
Limitações de uma implementação 2
Complexidade Computacional 4
PILHAS 6
Exercícios sobre Pilhas 11
FILAS 12
LISTAS 15
Exercícios Sobre Listas, Filas e Pilhas 18
Apontadores ou Ponteiros 19
Pilhas utilizando apontadores 20
Árvores 23
Caminhamento em Árvores 24
Árvores Binárias 25
Árvores Binárias de Busca 26
Implementação de uma Árvore Binária de Busca em C# 28
Listas Simplesmente Encadeadas 31
Listas Duplamente Encadeadas 31
Listas circulares 32
Grafos 33
Recursividade ou Recursão 34
Ordenação 36
Pesquisa em Memória Primária 40

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.

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 1.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000 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:

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

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.

Referências:

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

Complexidade Computacional

Fonte: http://www.dca.fee.unicamp.br/~ting/Courses/ea869/faq1.html

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.

```
{encontrando o maior número}
maior:= numero[1];   1 operação
for contador := 2 to 10 do  // n operações
  if numero[contador] > maior then  // 1 operação
    maior := numero[contador];  // 1 operação
```

A complexidade é dada por **n** elementos (o elemento que determina a taxa de crescimento do algoritmo acima) ou **O(n)** **(**linear**)**

```
For T:=1 to 100 do  // n operações
     For X := 1 to 100 do  // n operações
          Writeln(T, X);  // 1 operação
```

A complexidade é dada por $\mathbf{n^2}$ **(T*X)** elementos ou $\mathbf{O(n^2)}$ (quadrádico)

Funções limitantes superiores mais conhecidas:

Melhor → Pior

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

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	$1{,}79 \times 10^{308}$

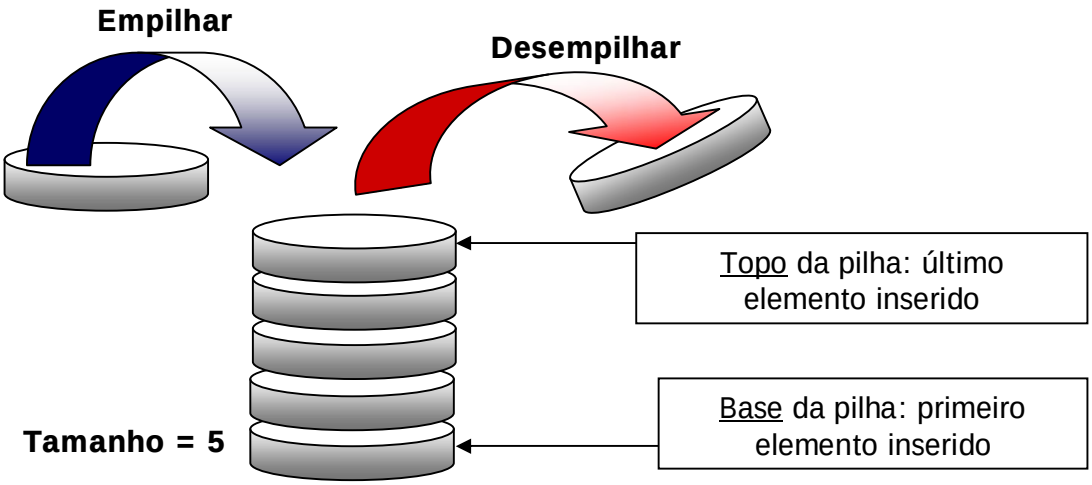
PILHAS

(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:

- **TOP (topo)**: acessa-se o elemento posicionado no topo da pilha;
- **PUSH (empilhar)**: insere um novo elemento no topo da lista;
- **POP (desempilhar)**: remove o elemento do topo da lista.
- **ISEmpty (vazia?)**: indica sea pilha está vazia.
- **Size (tamanho):** 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		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

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

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

namespace PilhaEstatica
{
    // definição da classe Pilha
    public class Pilha
    {
        private const int CAPACIDADE = 10; //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 retorna true se a pilha estiver vazia
        public bool vazia()
        {
            return tamanho() == 0;
        }

        // este método informa o tamanho da pilha
        public int tamanho()
        {
            return topo + 1;
        }

        // este método empilha um valor string na pilha
        public void empilha(string p_valor)
        {
            if (tamanho() == CAPACIDADE)
            {
                topo++;
                dados[topo] = p_valor;
            }
            else
            {
                Console.WriteLine("A PILHA ESTA CHEIA!!!");
            }
        }

        // este método desempilha um valor da pilha
        public string desempilha()
        {
            if (vazia() == true)
            {
                Console.WriteLine("A pilha está vazia!!!");
                return "";
            }
            else
            {
                topo--;
                return dados[topo + 1];
            }
        }

        // este método devolve o valor que está no topo
        public string retornatopo()
        {
            if (vazia() == true)
            {
                Console.WriteLine("A pilha está vazia!!!");
                return "";
            }
            else
            {
                return dados[topo];
            }
        }
    }
```

```
  // classe do programa principal.
    class Pilha_Vetor
    {
        static void Main(string[] args)
        {
            int opcao;
            string valor;
            Pilha  minhaPilha = new Pilha(); // cria uma instância da classe pilha!

            do
            {
                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
                }
            }
            while (opcao != 9);
        }
    }


}
```

```
// Esta pilha armazena em cada posição da pilha um dado heterogêneo( código e nome)

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

namespace PilhaEstatica
{
    // Estrura para armazenar dados heterogêneos no vetor.
    public struct registro
    {
        public int codigo;
        public string nome;
    };

    // definição da classe Pilha
    public class Pilha
    {
        private const int CAPACIDADE = 10; //define o tamanho maximo desta uma pilha.
        private registro[] dados = new registro[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 retorna true se a pilha estiver vazia
        public bool vazia()
        {
            return tamanho() == 0;
        }

        // este método informa o tamanho da pilha
        public int tamanho()
        {
            return topo + 1;
        }

        // este método empilha um valor string na pilha
        public void empilha(registro p_valor)
        {
            if (tamanho() == CAPACIDADE)
            {
                topo++;
                dados[topo] = p_valor;
            }
            else
            {
                Console.WriteLine("A PILHA ESTA CHEIA!!!");
            }
        }

        // este método desempilha um valor da pilha
        public registro desempilha()
        {
            if (vazia() == true)
            {
                Console.WriteLine("A pilha está vazia!!!");

                registro nada;
                nada.codigo = 0;
                nada.nome   = "";
                return nada;
            }
            else
            {
                topo--;
                return dados[topo + 1];
            }
        }

        // este método devolve o valor que está no topo
        public registro retornatopo()
        {
            if (vazia() == true)
            {
```

```
                Console.WriteLine("A pilha está vazia!!!");
                registro nada;
                nada.codigo = 0;
                nada.nome = "";
                return nada;
            }
            else
            {
                return dados[topo];
            }
        }
    }


    // classe do programa principal.
    class Pilha_Vetor
    {
        static void Main(string[] args)
        {
            int opcao;
            registro dado;
            Pilha  minhaPilha = new Pilha(); // cria uma instância da classe pilha!

            do
            {
                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 nome para empilhar: ");
                    dado.nome = Console.ReadLine();
                    Console.Write(">>Digite o código para empilhar: ");
                    dado.codigo = Convert.ToInt32( Console.ReadLine() );

                    minhaPilha.empilha(dado);
                }
                else if (opcao == 2)
                {
                    dado = minhaPilha.desempilha();
                    Console.WriteLine(">>Desempilhado: Código: {0}  Nome: {1} \n\n",
                                      dado.codigo, dado.nome);
                }
                else if (opcao == 3)
                {
                    dado = minhaPilha.retornatopo();
                    Console.WriteLine(">>Dado no topo: Código: {0}  Nome: {1} \n\n",
                                      dado.codigo, dado.nome);
                }
                else if (opcao == 4)
                {
                    Console.WriteLine(">>Tamanho da pilha: {0}", minhaPilha.tamanho());
                }
                else if (opcao == 9)
                {
                    // sai do programa
                }
            }
            while (opcao != 9);
        }
    }


}
```

Exercícios sobre Pilhas

1. Crie uma pilha que manipule a seguinte estrutura:

```
Tfuncionario  = registro
   Nome : string;
   Salario : Real;
end;
```

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

A pilha deve possuir os seguintes métodos:

- Empilhar (p_funcionario); ⇒ Empilhar um dado do tipo da estrutura que você definir.
- Desempilhar : Tfuncionario; ⇒ Desempilhar um valor e retornar o valor desempilhado
- RetornaTopo : Tfuncionario; ⇒ Retorna o valor que está no topo da pilha
- Tamanho : integer; ⇒ Retorna o tamanho da pilha
- Listar ⇒ Exibe na tela os elementos da pilha, ou exibe "pilha vazia".
- SomaSalarios : real ⇒ Retorna a soma de todos os salários de todos os funcionários.

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

2. Preencha a tabela abaixo, de acordo com os métodos executados na primeira coluna:

Operação	Pilha (topo) ----- (base)	Retorno	Tamanho da Pilha
Tamanho			
Empilhar('O')			
Empilhar('Z')			
RetornaTopo			
Empilhar('O')			
Vazio?			
Desempilhar			
Tamanho			
Empilhar('X')			
RetornaTopo			
Desempilhar			
Empilhar('O')			
Empilhar('B')			

Veja o exemplo de preenchimento na documentação entregue sobre **Pilhas**.

FILAS

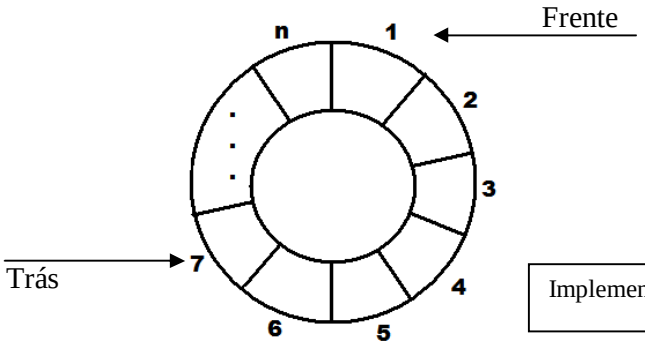
Uma Fila (*Queue*) é um caso particular de Listas Lineares que obedece ao critério FIFO (*First In, Firts Out*, ou Primeiro a Entrar, Primeiro a Sair). Numa fila os elementos são inseridos em uma das extremidades e retirados na outra extremidade. Existe uma ordem linear para a fila que é a "ordem de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

Uma pilha 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. Consequentemente 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 trás ---→ frente	Retorno	Tamanho da Fila
Tamanho		0	0
Enfileirar('A')	A		1
Enfileirar('B')	BA		2
Vazia	BA	False	2
Enfileirar('C')	CBA		3
RetornaFrente	CBA	A	3
RetornaTras	CBA	C	3
Desenfilera	CB	A	2
Desenfilera	C	B	1
Desenfilera		C	0
Vazia		True	0
RetornaFrente		'ERRO'	0

```
// Implementação de uma fila circular em C# utilizando vetor

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


namespace FilaEstatica
{

class Fila{
    const int CAPACIDADE = 10;  // 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]; // este vetor irá 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)
        {
            Console.WriteLine("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)
      {
        Console.WriteLine("A fila está vazia!");
        return "";
      }
      else
      {
        string valor = dados[inicio];
        inicio = (inicio + 1) % CAPACIDADE;
        quantidade --;
        return valor;
      }
    }
} // fim da classe Fila



// programa principal para testar a fila circular.
class FilaEstatica
{
    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
        {
            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;
            }

        }while (opcao != "9");
    }
} // fim da classe FilaEstatica
```

LISTAS

http://www.inf.ufsc.br/~ine5384-hp/Capitulo2/EstruturasLista.html

Uma Estrutura de Dados **Lista** é um conjunto de dados dispostos e/ou acessáveis em uma seqüência determinada.

- Este conjunto de dados pode possuir uma ordem intrínseca (Lista Ordenada) ou não.
- Este conjunto de dados pode ocupar espaços de memória fisicamente consecutivos, espelhando a sua ordem, ou não.
- Se os dados estiverem dispersos fisicamente, 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 lineares em uma lista única.
- 6. Partir uma lista linear em duas ou mais listas.
- 7. Fazer uma cópia da lista linear.
- 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.

IMPLEMENTAÇÃO DE LISTAS LINEARES 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.

Abaixo, listagem de uma implementação de Lista com Arranjos:

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

namespace ListaEstatica
{

    public 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)
            {
                Console.WriteLine("A lista está cheia!!!\n\n");
            }
            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)
            {
                Console.WriteLine("A lista está vazia!!!!");
                return "";
            }
            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]);
            }
        }
    } // fim da classe lista


    class Program_ListaEstatica
    {
        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
            {
                Console.WriteLine("\nDigite: \n 1-> Inserir no início \n 2-> Inserir no fim  \n" +
                    "3-> Inserir em uma posição  \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;
                }
            } while (opcao != "9");
        }
    }
}
```

Exercícios Sobre Listas, Filas e Pilhas

Sabendo-se que é possível implementar uma Pilha e uma Fila utilizando a estrutura de dados **LISTA, faça:**

Exercício 1:

Dada a Estrutura de Dados LISTA descrita e implementada acima, faça o programa principal de forma que ele simule uma **Pilha** para armazenamento de strings**.**

Deve executar os métodos: **Empilhar, Desempilhar e Tamanho.**

Exercício 2:

Dada a Estrutura de Dados LISTA descrita e implementada acima, faça o programa principal de forma que ele simule uma **Fila** para armazenamento de strings**.**

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

Apontadores ou Ponteiros

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

Um apontador 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**

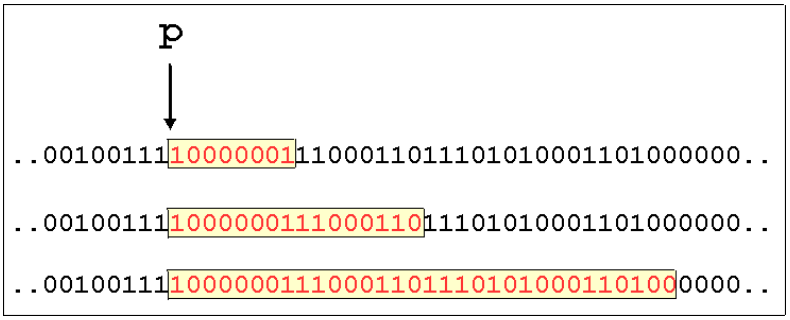
Variável Inteira **I**:

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

```
Writeln( *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.



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.

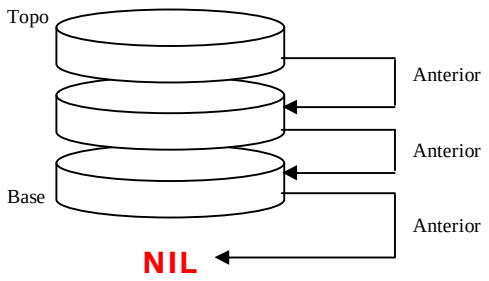
A esquerda temos 3 tipos de dados em Pascal: Byte na primeira linha, word na segunda e longint na terceira. Veja que os tamanhos são diferentes!

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 utilizando apontadores

As pilhas criadas utilizando-se apontadores são mais eficientes, pois não precisamos especificar o tamanho inicial, tampouco é necessário definir um Limite. Cada elemento da pilha aponta para o próximo, 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 **NIL.**

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

```
program  Ppilha;
Type
  Apontador = ^TElemento;

  TElemento = record
    Anterior : apontador;
    dado : String;
```

```
end;

Pilha = Object
      Topo : Apontador;
      Procedure Empilha( valor : string );
      ...
```

```
{pilha utilizando apontadores}
program  Ppilha;
uses crt;
type
   {tipo de dado Apontador para apontar para
    um elemento da pilha}
   Apontador = ^TElemento;

  {estrutura que representa um elemento na pilha}
   TElemento = record
      anterior : apontador;
      dado     : String;
   end;

   {estrutura que representa uma pilha}
   Pilha = Object
       Topo : Apontador;
       quantidade : integer;

       procedure inicializa;
       function  Tamanho : integer;
       procedure Listar;
       procedure empilha( p_valor : String );
       function  desempilha : string;
   end;

  {inicia uma pilha vazia}
  procedure Pilha.inicializa;
  begin
      topo  := nil;
      quantidade := 0;
  end;


  {retorna o tamanho da pilha}
  function Pilha.Tamanho : integer;
  begin
     Tamanho := quantidade;
  end;


  {lista os elementos da pilha}
  procedure Pilha.Listar;
  var elemento : Apontador;
  begin
     if tamanho > 0  then
     begin
        writeln('Informa‡äes contidas na pilha');
        elemento := topo;
        while elemento <> NIL do
        begin
           Writeln( elemento^.dado );
           elemento := elemento^.anterior;
        end;
     end
     else
     begin
        writeln('A pilha esta vazia!!!');
     end;
  end;


  {empilha um valor string}
  procedure Pilha.empilha( p_valor : String );
  var elemento : apontador;
  Begin
     {reserva mem¢ria para o novo elemento}
     new (elemento);

     elemento^.anterior := topo;
     elemento^.dado     := p_valor;


     if tamanho > 0 then
     begin
        elemento^.anterior := topo;
     end;
     topo := elemento;
     quantidade := quantidade + 1;
  end;


  {desempilha um valor string}
  function Pilha.desempilha : string;
  var elemento : apontador;
  Begin
     if (tamanho = 0) then
     begin
        desempilha := '';
        writeln('A piloha esta vazia!');
     end
     else
     begin
        desempilha := topo^.dado;
        {guarda p/ depois liberar o espaco}
        elemento   := topo;
        topo := topo^.anterior;
        dispose(elemento);
        quantidade := quantidade  - 1;
     end;
  end;


{programa principal para testar a pilha}
var
  MinhaPilha : Pilha;
  valor : string;
  tecla : char;
begin
  clrscr;
  MinhaPilha.inicializa;

  repeat
    writeln;
    writeln('Informe 1->Empilhar 2->Desempilhar '
    + '3->Tamanho 4->Listar e 5->Sair');
    readln(tecla);

    if (tecla = '1') then
    begin
      write('Informe algo p/ empilhar: ');
      readln(valor);
      MinhaPilha.empilha( valor );
    end

    else if (tecla = '2') then
    begin
      writeln('Desempilhado: ' ,
```

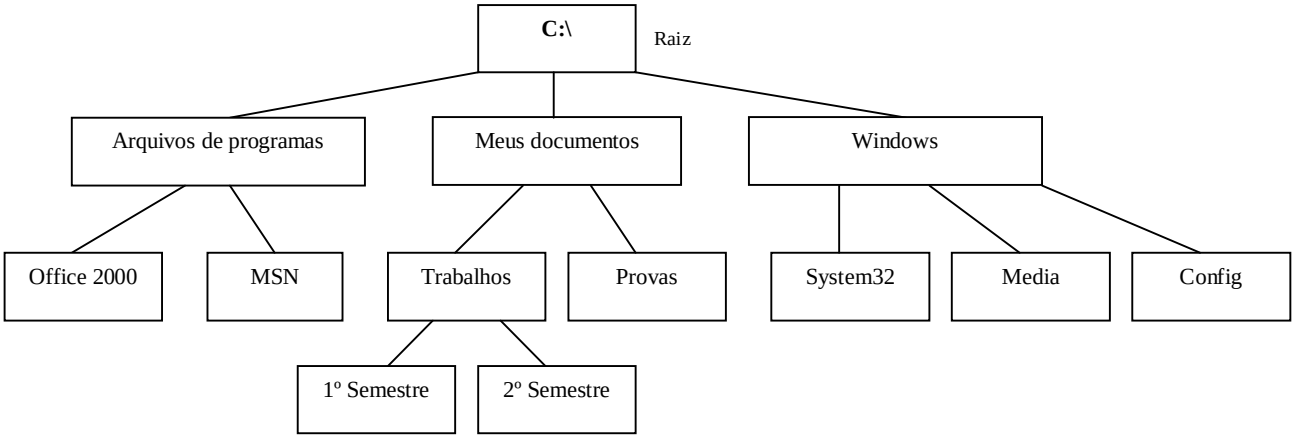
```
          minhaPilha.Desempilha );
end
else if (tecla = '3') then
begin
  writeln('Tamanho: ' , MinhaPilha.Tamanho );
end
else if (tecla = '4') then
begin
   MinhaPilha.Listar;
    end;
  until (tecla = '5');
end.
```

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).

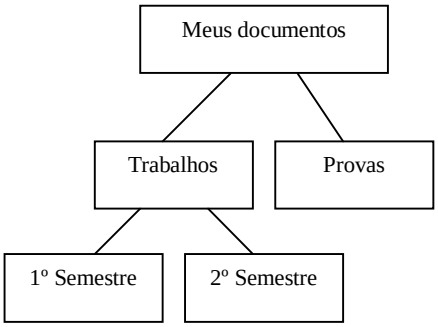
EX: Uma árvore que representa a estrutura de pastas em um Hard Disk:



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.



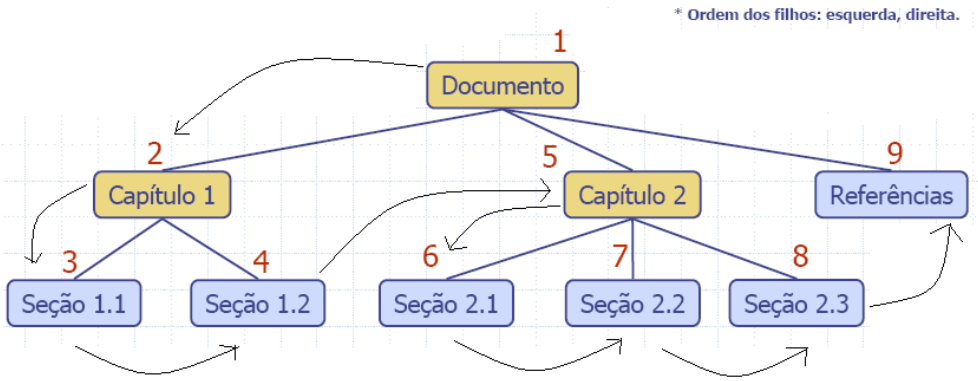
Os principais métodos de uma árvore são:

- **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.

Caminhamento em Árvores

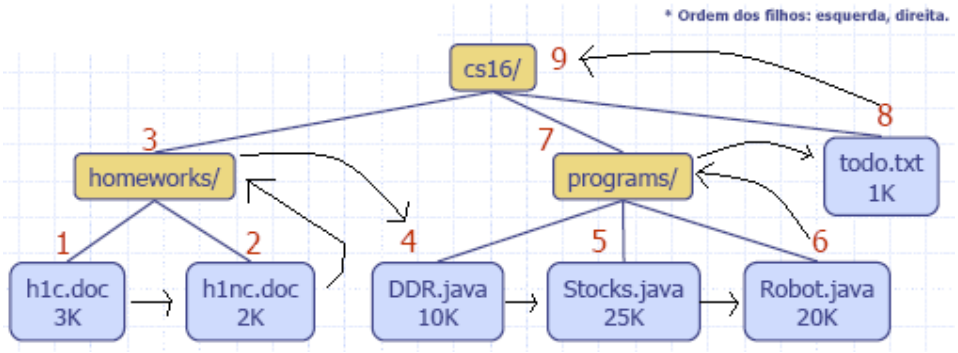
O caminhamento de uma árvore é a maneira ordenada de percorrer todos os nodos da árvore (percorrer todos os seus nós, sem repetir nenhum e sem deixar de passar por nenhum). É utilizada, por exemplo, para consultar ou alterar as informações contidas nos nós.

Caminhamento 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.

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



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].

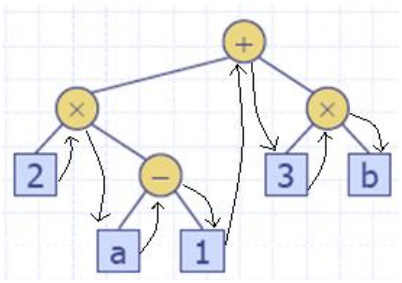
Uma **árvore binária** é uma árvore ordenada na qual todo nodo **tem, no máximo, dois filhos**. Uma árvore binária imprópria é aquela que possui apenas 1 filho. Já uma árvore binária própria é aquela em que todo nodo tem zero ou dois filhos, ou seja, todo nodo interno tem exatamente 2 filhos. Isso porque 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**.

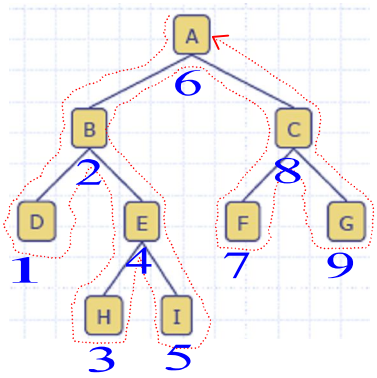


Os elementos acessados por este caminhamento formam a expressão:

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.



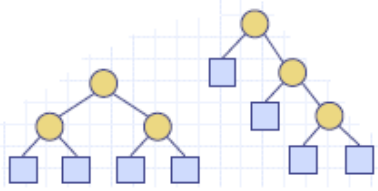
Prioridade das ações para efetuar o caminhamento:

- 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.



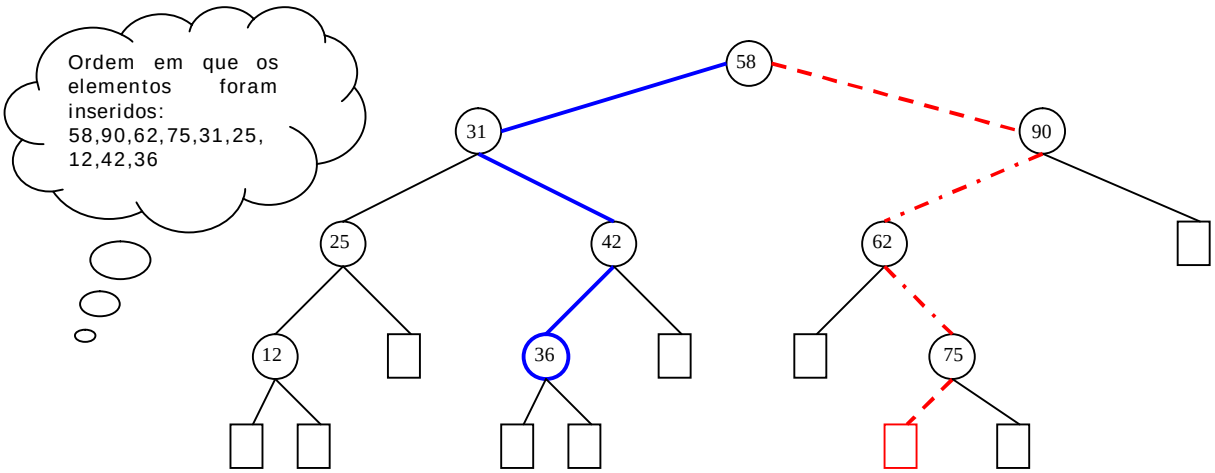
Árvores Binárias de Busca

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.

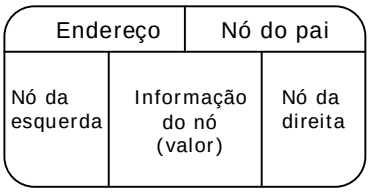
Podemos usar uma árvore binária de pesquisa **T** para localizar um elemento com um certo valor **x** percorrendo para baixo a árvore **T**. Em cada nodo interno, comparamos o valor do nodo corrente com o valor do elemento **x** sendo pesquisado.

- 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.



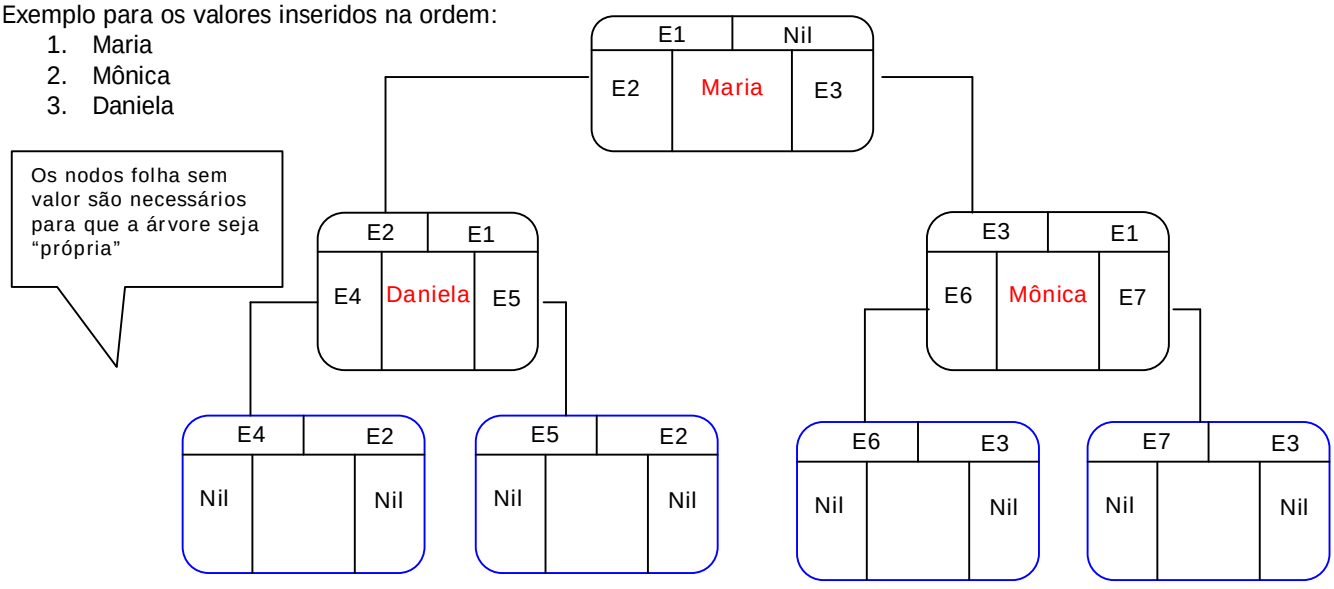
A figura acima representa uma árvore binária de pesquisa que armazena inteiros. O caminho indicado pela linha azul corresponde ao caminhamento ao procurar (com sucesso) 36. A linha pontilhada vermelha corresponde ao caminhamento ao procurar (sem sucesso) por 70. **Observe que o tempo de execução da pesquisa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



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

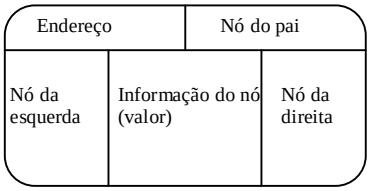
```
Pesquisa(  nodo, valor_pesquisado ) : Retorno
Inicio
        se Nodo_eh_externo(Nodo) = verdadeiro então
                escreva( 'Erro: Valor procurado não está na árvore!');
                pesquisa := nil
        caso contrário
                Se valor_pesquisado < nodo.valor então
          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

Implementação de uma Árvore Binária de Busca em C#

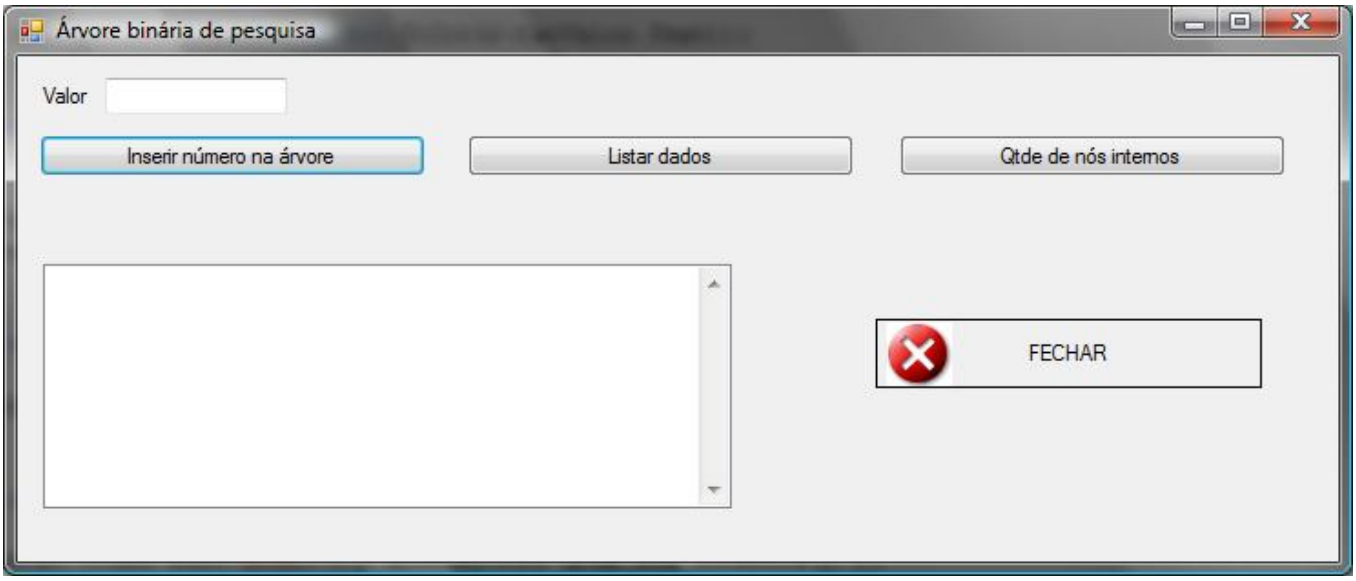



```
// classe para reprentar 1 Nodo na árvore
    class Nodo
    {
        private Nodo no_pai = null;
        private Nodo no_direita = null;
        private Nodo no_esquerda = null;
        private int valor = 0;


        public int get_valor()       { return valor; }

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

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

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

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

        public Nodo get_no_pai() { return no_pai; }

        public Nodo get_no_direita() { return no_direita; }

        public Nodo get_no_esquerda()  { return no_esquerda; }
   }
```

```
// classe da árvore de pesquisa binária
    class ArvoreBin
    {
        private Nodo raiz = null; // raiz da árvore
        private int qtde = 0; // qtde de nos internos
        private string resultado = "";


        public int qtde_nos_internos() // devolve a qtde de nós internos
        {
            return qtde;
        }

        public bool no_eh_externo(Nodo no) // verifica se um determinado Nodo é externo
        {
            return (no.get_no_direita() == null) && (no.get_no_esquerda() == null);
        }


        public Nodo cria_No_externo(Nodo Nopai)// cria um Nodo externo
        {
            Nodo no = new Nodo();
            no.set_no_pai(Nopai);
            return no;
        }
```

```
        public void insere(int valor) // insere um valor int
        {
            Nodo no_aux;

            if (qtde == 0)
            {
                // 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 = raiz;
                while (no_eh_externo(no_aux) == false)
                {
                    if (valor > no_aux.get_valor())
                        no_aux = no_aux.get_no_direita();
                    else
                        no_aux = no_aux.get_no_esquerda();
                }
            }
            // este era um Nodo externo e portanto não tinha filhos.
            // Agora ele passará a ter valor. Também devemos criar outros 2
            // Nodos externos (filhos) para ele.
            no_aux.set_valor(valor);
            no_aux.set_no_direita(cria_No_externo(no_aux));
            no_aux.set_no_esquerda(cria_No_externo(no_aux));
            qtde++;
        }

        private void Le_Nodo(Nodo no)
        {
            if (no_eh_externo(no))
                return;

            Le_Nodo(no.get_no_esquerda());
            resultado = resultado + " - " + Convert.ToInt32(no.get_valor());
            Le_Nodo(no.get_no_direita());
        }

        // devolve um string com os elementos da árvore, em ordem crescente
        public string listagem()
        {
            resultado = "";
            Le_Nodo(raiz);
            return resultado;
        }
    }
```

Interface com o Usuário:



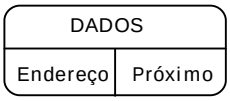
Código da interface com o usuário:

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

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                minhaArvore.insere(Convert.ToInt32(txtValor.Text));
                listBox1.Items.Add("Inserido: " + txtValor.Text);
            }
            catch{
                MessageBox.Show("Valor inválido! Digite apenas números!");
            }
            txtValor.Clear();
            txtValor.Focus();
        }

        private void button2_Click(object sender, EventArgs e)
        {
            listBox1.Items.Add(minhaArvore.listagem());
        }

        private void button3_Click(object sender, EventArgs e)
        {
            listBox1.Items.Add("Qtde: " + minhaArvore.qtde_nos_internos() );
        }

        private void button4_Click(object sender, EventArgs e)
        {
            Close();
        }
  }
```

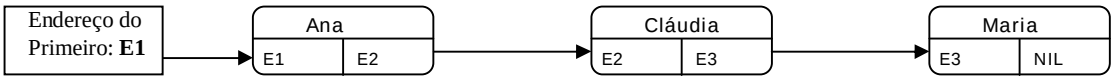
Listas Simplesmente Encadeadas

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

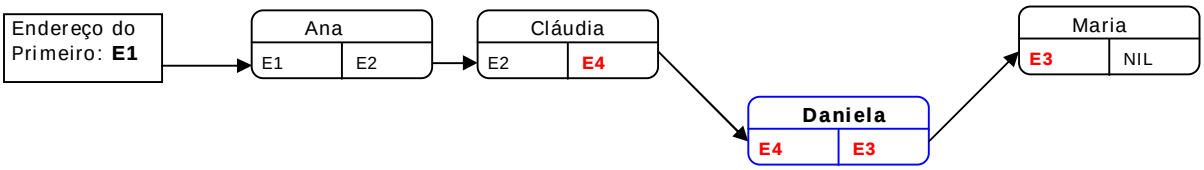
Em uma lista simplesmente encadeada, cada elemento contém um apontador que aponta para a o elemento seguinte. Na implementação de listas utilizando vetores, os dados ocupavam posições contíguas da memória. Sendo assim, sempre que incluímos ou apagamos um elemento no meio da lista, precisamos reorganizar 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:

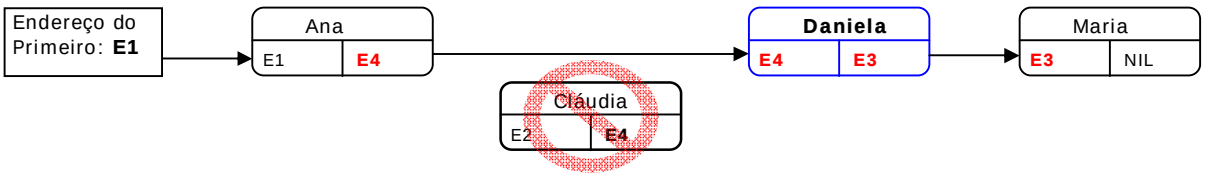


Para **incluir** um novo elemento, por exemplo, o nome Daniela, devemos apenas alterar o apontador próximo do elemento que está no endereço E2. Veja abaixo:



Observe que a ordem dos endereços não importa. O que importa é a ordem que eles estão encadeados!

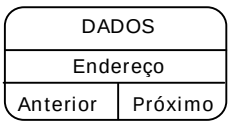
O mesmo ocorre ao se **remover** um elemento da lista. Veja abaixo como ficaria a remoção do elemento Cláudia:



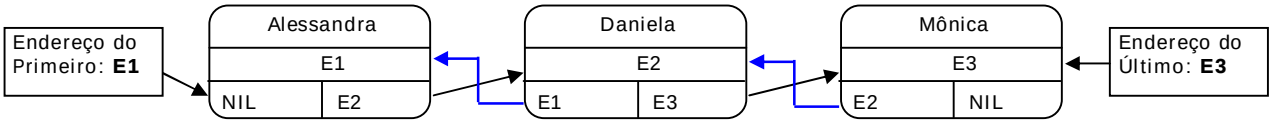
Listas Duplamente Encadeadas

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:

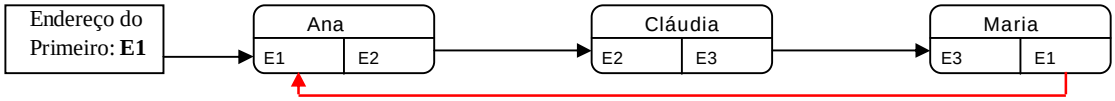


Para **Inserir** e **Remover** elementos, o processo é semelhante ao apresentado na lista simplesmente encadeada. A diferença é que na lista duplamente encadeada é necessário também atualizar o campo “anterior” dos elementos.

Listas circulares

Material retirado da referência [4].

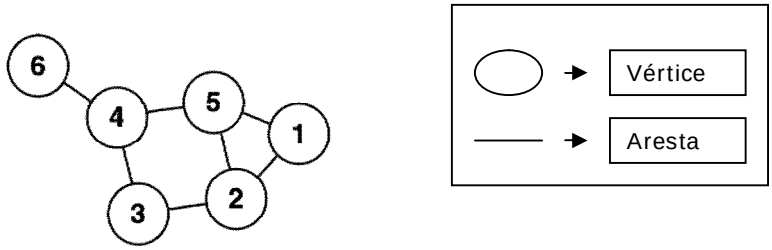
São listas que possuem a característica especial de ter, como sucessor do fim da lista, seu início, ou melhor, o fim da lista “aponta” para seu início, formando um círculo que permite uma trajetória contínua na lista. Veja o processo na ilustração abaixo:



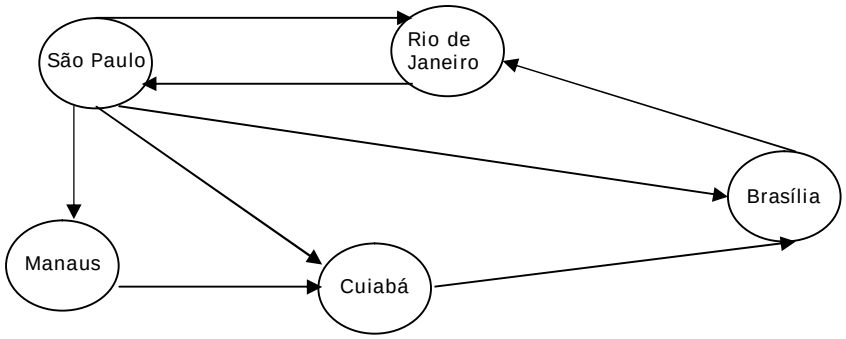
Grafos

Material sobre grafos: [3], [4] e http://www.inf.ufsc.br/grafos/livro.html

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**, ou dígrafo. 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 setas que entram ou saem de um nó.
- **Grau de entrada**: número de setas que chegam em um nó X, in(X).
- **Grau de saída**: número de setas que saem de um nó X, out(X).
- **Fonte**: todo nó, cujo grau de entrada é 0(zero).
- **Sumidouro (poço)**: todo nó, cujo grau de saída é 0(zero).

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.

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).

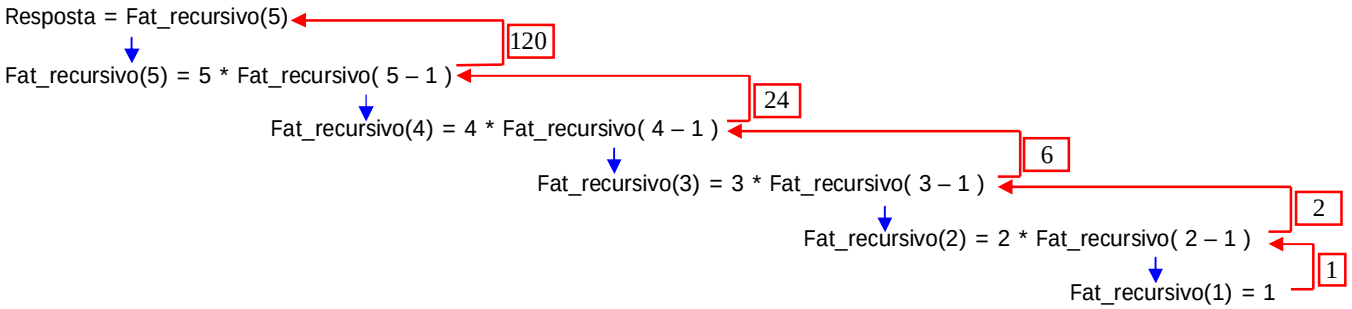
Um outro exemplo simples poderia ser o seguinte:

Se uma palavra desconhecida é vista em um livro, o leitor pode tomar nota do número da página e colocar em uma pilha (que até então está vazia). O leitor pode consultar esta nova palavra e, enquanto lê o texto, pode achar mais palavras desconhecidas e acrescentar no topo da pilha. O número da página em que estas palavras ocorrem também são colocados no topo da pilha. Em algum momento do texto, o leitor vai achar uma frase ou um parágrafo onde está a última palavra anotada e pelo contexto da frase vai descobrir o seu significado. Então o leitor volta para a página anterior e continua lendo dali. Paulatinamente, remove-se seqüencialmente cada anotação que está no topo da pilha. Finalmente, o leitor volta para a sua leitura já sabendo o significado da(s) palavra(s) desconhecida(s). Isto é uma forma de recursão.

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!

Em cauda: As funções recursivas em cauda formam uma subclasse das funções recursivas, nas quais a chamada recursiva é a última instrução a ser executada. Por exemplo, a função a seguir, para localizar um valor em uma lista ligada é recursiva em cauda, por que a última coisa que ela faz é invocar a si mesma:

Ex: Vamos usar como exemplo o 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 := nil
        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
```

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.

Ordenação

Material retirado de:

Referência [2], [3]

http://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o (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 vector 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 (n2) / 2 operações relevantes. No pior caso, são feitas 2n2operações. No caso médio, são feitas (5n2) / 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 VECTOR de elementos que podem ser comparados e n é o tamanho desse vector.

```
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] vem depois de 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);

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

```
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");
            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.

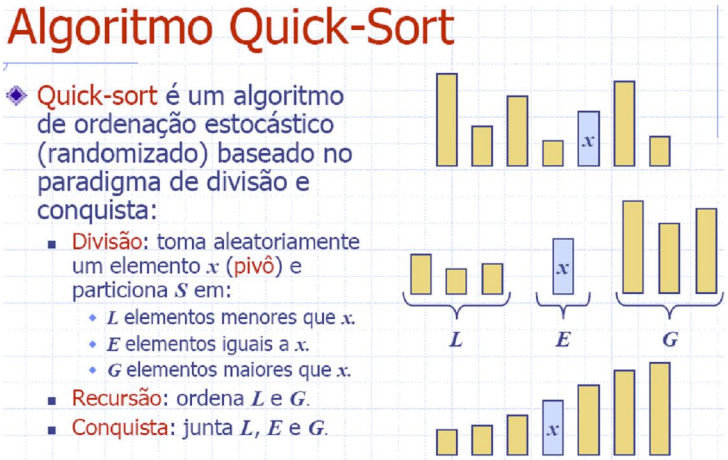
Complexidade

- $O(n \lg_2 n)$ no melhor caso e no caso médio
- $O(n^2)$ no pior caso;

Implementações

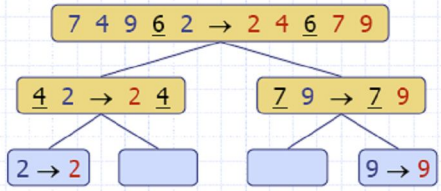
Algoritmo em português estruturado

```
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

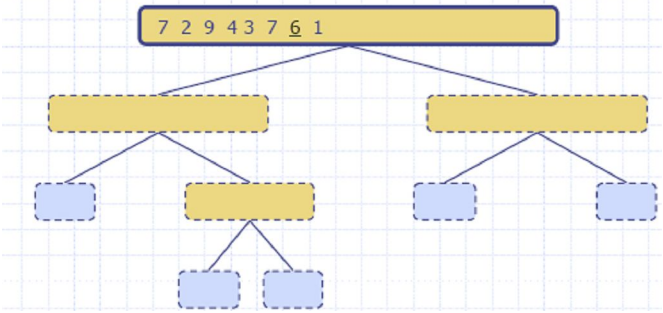
- Uma execução do quick-sort pode ser representada através de uma árvore binária:
  - A raiz representa a chamada inicial.
  - Os demais nós representam chamadas recursivas do algoritmo.
  - As folhas representam chamadas para subseqüências de tamanho 0 ou 1.



33

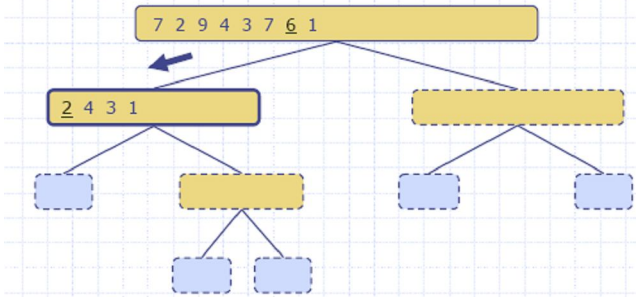
Exemplo de Execução

- Seleção do pivô:



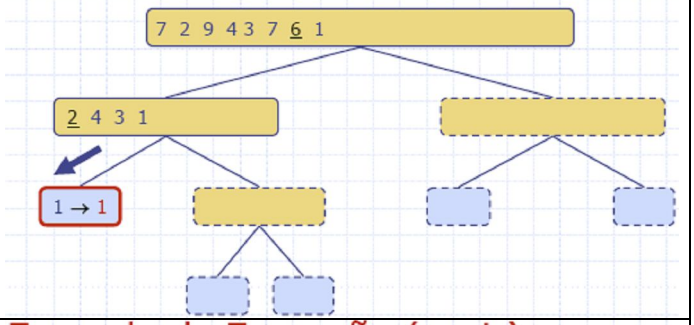
Exemplo de Execução (cont.)

- Partição, chamada recursiva e seleção do pivô:



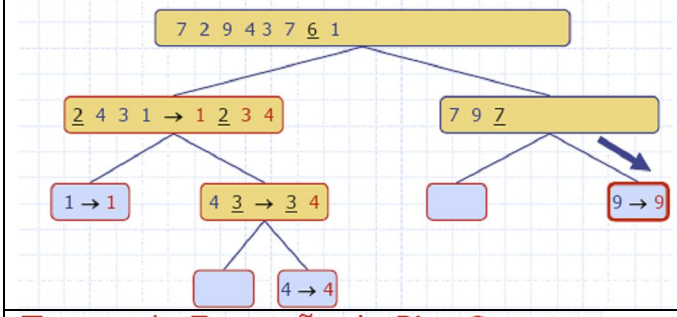
Exemplo de Execução (cont.)

- Partição, chamada recursiva e caso básico:



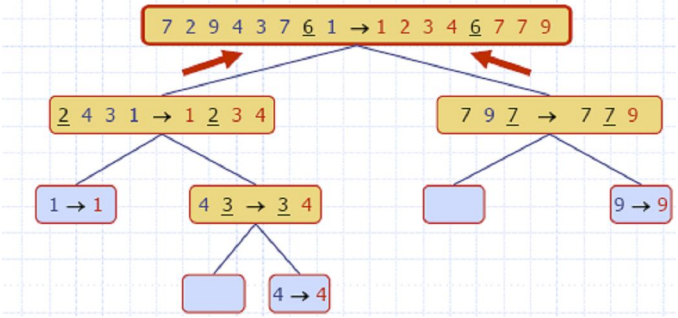
Exemplo de Execução (cont.)

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



Exemplo de Execução (cont.)

- Junção, junção

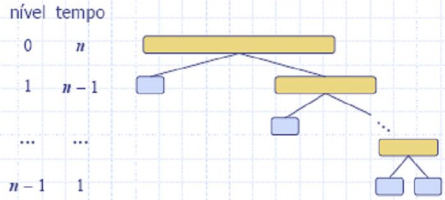


Tempo de Execução do Pior Caso

- O pior caso para o quick-sort ocorre quando o pivô é o único mínimo ou máximo elemento da seqüência.
- Nesse caso, ou L ou G possui tamanho $n-1$, enquanto o outro possui tamanho 0.
- O tempo de execução é portanto proporcional a:

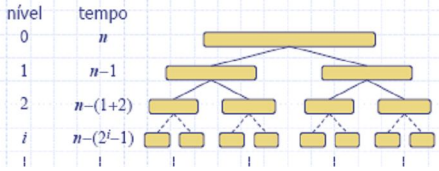
$$n + (n-1) + \ldots + 2 + 1$$

- Logo, quick-sort is $O(n^2)$.



Tempo Esperado e do Melhor Caso

- O melhor caso para o quick-sort ocorre quando o pivô é tal que a seqüência S seja dividida em L e G de tamanhos ≈ iguais.
- Nesse caso, é possível demonstrar que a altura da árvore é $O(\log n)$.
- O tempo de execução em uma dada profundidade i (fase de divisão) é proporcional a n menos uma constante, isto é, $O(n)$.
- Logo, quick-sort é $O(n \log n)$ no melhor caso, isto é, $\Omega(n \log n)$.
- Problema é que não podemos descobrir o pivô ideal sem inspeção.
- Seleção uniformemente aleatória do pivô garante tempo esperado também proporcional a $n \log n$, isto é, $O(n \log n)$ em média.



42

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();
        }
```

Pesquisa em Memória Primária

Pesquisa seqüencial

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 log2 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.