ESTRUTURAS DE DADOS

(Recursividade, Ordenação, Listas Lineares e Árvores)

Prof. Camillo Falcão

Sumário

APRESENTAÇÃO

Estrutura de Dados é um tema comumente visto junto a algoritmos. Você mesmo já aprendeu, na disciplina Fundamentos de Programação, algumas estruturas de dados como os vetores e matrizes. Nessa disciplina trataremos de algumas estruturas de dados que são fundamentais no desenvolvimento de programas, sendo que várias dessas estruturas estarão sendo apresentadas a você pela primeira vez.

Esta disciplina tem como objetivos de ensino:
- Apresentar diferentes abordagens de algoritmos de ordenação e comparar o seu desempenho em diferentes tamanhos de vetores.
- Explicar algoritmos recursivos e comparar o desempenho de diferentes abordagens recursivas e suas versões iterativas.
- Apresentar diferentes estruturas de dados, exemplificando a sua utilização.
- Demonstrar as vantagens da utilização de uma busca binária em substituição a uma busca linear.
- Discutir sobre quais estruturas são mais eficientes para cada problema proposto.
- Classificar as diferentes estruturas como estáticas e dinâmicas, quanto ao gasto de memória e desempenho em diferentes tipos de aplicações.

Já os objetivos de aprendizagem da disciplina Estrutura de Dados são:
- Implementar algoritmos que utilizam estruturas de dados para solucionar problemas computacionais.
- Comparar abordagens recursivas e iterativas e escolher qual a mais adequada para o problema proposto, considerando a manutenibilidade do código fonte e o desempenho desejado.
- Interpretar algoritmos de ordenação e escolher o mais adequado para cada cenário.
- Escolher a estrutura de dados mais adequada para a solução de um problema computacional.

Esse material é composto de quatro unidades: Recursividade, Ordenação, Listas Encadeadas e Árvores. Nele serão apresentados alguns algoritmos. Lembre-se de sempre testar e analisar cada algoritmo apresentado.

Ao término de cada unidade teremos um conjunto de exercícios para fixação. Esses exercícios são fundamentais para a consolidação dos conteúdos apresentados.

Dica: nesse momento, você não deve se preocupar se a sua solução para um determinado exercício ficar muito grande. Isso é normal e ao longo do curso você terá a oportunidade de praticar a sua lógica de programação, o que fará com que o seu código fique cada vez mais simples.

UNIDADE I – INTRODUÇÃO À RECURSIVIDADE

Algumas tarefas em computação precisam ser executadas mais de uma vez. Você já aprendeu, na disciplina Fundamentos de Programação (Algoritmos), a utilizar laços para repetir um determinado trecho de código um certo número de vezes, ou repetir enquanto uma determinada condição for verdadeira.

Em nosso dia a dia, é comum utilizarmos repetição, como quando em receitas culinárias você segue a instrução “mexa até desgrudar do fundo da panela”.

As sub-rotinas são recursos muito importantes em computação. Com uma sub-rotina é possível nomear um determinado bloco de código, sendo que esse bloco pode receber dados de entrada e pode produzir dados de saída. Ao dar um nome para um bloco de código, criando com isso uma sub-rotina, é possível executar esse bloco de código somente chamando-o através do seu nome. Esse recurso é importante para a organização do código fonte de um programa.

Em nosso dia a dia, também é comum darmos nome a blocos de instruções, como o nome que damos para as nossas receitas culinárias. Veja a receita abaixo, retirada do site https://incrivel.club/inspiracao-criancas/6-receitas-faceis-para-preparar-com-as-criancas-35355/

Pudim de banana e chocolate

Ingredientes:	**Preparo:**
3 bananas. 4 colheres de creme de leite. 250g de creme de chocolate para untar	Descasque as bananas e, em um recipiente, triture-as com um garfo. Misture o creme de leite com o creme de chocolate. Adicione o 'purê' de banana à mistura de chocolate. Leve à geladeira por uma hora e depois aproveite com as crianças esta simples delícia.

Se imaginássemos que um robô prepararia a receita acima, poderíamos dividi-la em duas partes: separação dos ingredientes e preparo da receita. A nossa receita, estruturada de uma forma a lembrar uma sub-rotina, ficaria assim:

```
PrepararPudimDeBananaEChocolate()
{
      SepararIngredientes();
      PrepararReceita();
}

SepararIngredientes()
{
      Separe 3 bananas.
      Separe 4 colheres de creme de leite.
      Separe 250g de creme de chocolate.
}

PrepararReceita()
{
      Descasque as bananas.
      Em um recipiente, triture as bananas com um garfo.
      Misture o creme de leite com o creme de chocolate.
      Adicione o 'purê' de banana à mistura de chocolate.
      Leve à geladeira por uma hora.
}
```

Agora imagine que, por engano, a nossa primeira sub-rotina tivesse ficado com esse código:

```
PrepararPudimDeBananaEChocolate()
{
      SepararIngredientes();
      PrepararReceita();
      PrepararPudimDeBananaEChocolate();
}
```

Note que após separar os ingredientes e preparar a receita, é feita uma chamada à própria sub-rotina PrepararPudimDeBananaEChocolate. Isso faz com que uma nova separação de ingredientes e preparo da receita sejam feitos e, ao final, novamente se inicie um novo preparo. Com a sub-rotina acima, o robô ficaria em loop, repetindo a receita sem parar. Isso provavelmente causaria um erro, pois em algum momento, ao tentar separar os ingredientes, algum ingrediente poderia não mais estar disponível.

E se nós aproveitássemos essa sub-rotina alterada para fazermos o robô repetir a receita enquanto for possível fazê-lo? Uma forma simples de fazermos isso se encontra na seguinte versão modificada de nossa sub-rotina:

```
PudimDeBananaEChocolate()
{
    Se existem ingredientes suficientes
    {
      SepararIngredientes();
      PrepararReceita();
      PudimDeBananaEChocolate();
    }
}
```

Com essa nova versão da nossa sub-rotina, a repetição ocorrerá enquanto houverem ingredientes suficientes.

Essa sub-rotina que chama a si mesma é uma sub-rotina recursiva. A condição que permite que essa sub-rotina não seja chamada indefinidamente é conhecida como condição de parada.

Vamos contextualizar melhor o que é uma sub-rotina recursiva em computação?

Quando uma sub-rotina chama a si própria para atingir o resultado a que se propõe, essa é uma sub-rotina recursiva. Essa estratégia de solução de problema pode ser utilizada sempre que o cálculo de uma função para um determinado valor n pode ser descrito pelo cálculo dessa mesma função para o valor anterior a n (SZWARCFITER, MARKEZON, 2010).

A recursividade pode ser direta ou indireta. A recursividade direta ocorre quando uma sub-rotina A possui uma chamada para ela mesma. Já a recursividade indireta ocorre quando uma sub-rotina A chama uma sub-rotina B e esta sub-rotina B chama a sub-rotina A (CELES, CERQUEIRA, RANGEL, 2017).

Nesse ponto, podemos explorar algumas sub-rotinas recursivas. Faremos isso no próximo tópico.

TÓPICO I – ITERAÇÃO VERSUS RECURSIVIDADE

A multiplicação de inteiros positivos é um exemplo de um problema que pode ser resolvido recursivamente. Apesar de ser mais fácil a você utilizar o operador de multiplicação, vamos construir uma versão recursiva somente para lhe ajudar a entender como a recursividade pode ser utilizada.

A operação de multiplicação de inteiros positivos pode ser definida em termos da operação de adição:

$$m \; x \; n = \underbrace{m + \cdots + m}_{n \; vezes}$$

Dessa forma, calculando o resultado de 2 x 3 chagaremos ao mesmo resultado que calculando o resultado de 2 + 2 + 2.

Um algoritmo iterativo pode ser facilmente utilizado para fazer a multiplicação dessa maneira. Veja o seguinte abaixo:

```
public static int Multiplicar(int a, int b)
{
    int resultado = 0;
    for (int i = 0; i < b; i++)
        resultado += a;
    return resultado;
}

public static void Main(string[] args)
{
    Console.Write(Multiplicar(7, 3));
}
```

Já na forma recursiva, para valores de n e m não negativos, podemos implementar a multiplicação de um número n por um número m somando n com a multiplicação de n por $m - 1$. Dessa forma, teríamos:

- $n * m = n + (n * m - 1)$;
- $2 * 3 = 2 + (2 * 2)$;
- $2 * 2 = 2 + (2 * 1)$;
- $2 * 1 = 2 + (2 * 0)$;
- $2 * 0 = 2 + (2 * -1)$;

Note que é feita uma nova chamada à multiplicação, porém agora essa chamada é feita para resolver um subproblema que é parte do anterior.

Considerando o exposto, por indução matemática, podemos definir recursivamente a multiplicação números inteiros não negativos como a seguir:

- $m \times n = 0$ $\quad se\ n = 0$
- $m \times n = m + (m \times (n - 1))$ $\quad se\ n > 0$

Veja como poderia ficar a nova versão do algoritmo, sendo esta agora uma solução recursiva:

```
public static int Multiplicar(int m, int n)
{
    if (n == 0)
        return 0;
    else
        return m + Multiplicar(m, n - 1);
}
```

```
public static void Main(string[] args)
{
    Console.Write(Multiplicar(7, 5));
}
```

Apresentamos acima um exemplo de um algoritmo iterativo que foi codificado como um algoritmo recursivo. Vamos falar no próximo tópico sobre problemas que são definidos recursivamente.

TÓPICO II – PROBLEMAS DEFINIDOS RECURSIVAMENTE

Considerando um número natural n, a multiplicação de todos os inteiros positivos menores ou iguais a n é conhecida como fatorial de n e sua notação é $n!$.

Veja a definição da função fatorial:

$$n! = \prod_{k=1}^{n} k \qquad \forall n \in \mathbb{N}$$

Essa definição implica que $0! = 1$, pois o produto sem nenhum número ou produto vazio é igual a 1.

Veja como é calculado o fatorial de 5:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

O fatorial de um número natural pode ser calculado por um algoritmo iterativo, como o algoritmo abaixo:

```
static int FatorialIterativo(int n)
{
    int result = 1;
    for (int i = n; i >= 1; i--)
        result *= i;
    return result;
}
```

O nosso objetivo agora é chegarmos à definição recursiva de fatorial. Para isso, note que $4! = 4 \times 3 \times 2 \times 1$. Com isso, podemos concluir que $5! = 5 \times 4!$. Veja também:

$$5! = 5 \times 4!$$
$$4! = 4 \times 3!$$
$$3! = 3 \times 2!$$
$$2! = 2 \times 1!$$
$$1! = 1 \times 0!$$
$$0! = 1$$

Após chegar ao 0!, você pode voltar à linha de cima, calcular o valor de 1! e utilizar esse valor para calcular 2! e assim sucessivamente até chegar ao valor de 5!. Veja isso sendo feito:

$$5! = 5 \times 4! = 5 \times 24 = 120$$
$$4! = 4 \times 3! = 4 \times 6 = 24$$
$$3! = 3 \times 2! = 3 \times 2 = 6$$
$$2! = 2 \times 1! = 2 \times 1 = 2$$
$$1! = 1 \times 0! = 1 \times 1 = 1$$
$$0! = 1$$

Temos então o problema definido recursivamente:

$$n! = \begin{cases} 1, & se\ n=0 \\ n \times (n-1)!, & se\ n>0 \end{cases}$$

E esta é uma versão recursiva do nosso algoritmo para cálculo do fatorial de um número natural:

```
static int Fatorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * Fatorial(n - 1);
}
```

TÓPICO III – PILHA DE EXECUÇÃO

Considere o seguinte algoritmo:

```
static void Procedimento1()
{
    Console.WriteLine("1. Empilhou o procedimento 1");
    Console.WriteLine("2. Procedimento 1 processando...");
    Procedimento2();
    Console.WriteLine("3. Procedimento 1 terminou o seu processamento.");
    Console.WriteLine("4. Desempilhou o procedimento 1");
}

static void Procedimento2()
{
    Console.WriteLine("  2.1. Empilhou o procedimento 2");
    Console.WriteLine("  2.2. Procedimento 2 processando...");
    Procedimento3();
    Console.WriteLine("  2.3. Procedimento 2 terminou o seu "
                      + "processamento.");
    Console.WriteLine("  2.4. Desempilhou o procedimento 2");
}
```

```
static void Procedimento3()
{
    Console.WriteLine("    2.2.1. Empilhou o procedimento 3");
    Console.WriteLine("    2.2.2. Procedimento 3 processando...");
    Console.WriteLine("    2.2.3. Procedimento 3 terminou o seu"
                      + "processamento.");
    Console.WriteLine("    2.2.4. Desempilhou o procedimento 3");
}

static void Main(string[] args)
{
    Procedimento1();
}
```

Nesse algoritmo, o *Procedimento1* chama o *Procedimento2* que, por sua vez, chama o *Procedimento3*. A saída para esse algoritmo é:

```
1. Empilhou o procedimento 1
2. Procedimento 1 processando...
  2.1. Empilhou o procedimento 2
  2.2. Procedimento 2 processando...
    2.2.1. Empilhou o procedimento 3
    2.2.2. Procedimento 3 processando...
    2.2.3. Procedimento 3 terminou o seu processamento.
    2.2.4. Desempilhou o procedimento 3
  2.3. Procedimento 2 terminou o seu processamento.
  2.4. Desempilhou o procedimento 2
3. Procedimento 1 terminou o seu processamento.
4. Desempilhou o procedimento 1
```

Note na saída do programa os momentos em que cada procedimento é empilhado e desempilhado.

Quando uma sub-rotina é chamada para a execução, é criado um registro de ativação na pilha de execução do programa. Nesse registro são armazenados os parâmetros, as variáveis locais e o ponto de retorno do programa ou sub-rotina que o chamou. Ao final da execução da sub-rotina, a mesma é desempilhada e a execução volta ao ponto de retorno da sub-rotina ou programa que o chamou (ASCENCIO, CAMPOS, 2012).

Pense em uma pilha de pratos. O primeiro prato a ser retirado da pilha é o que está no topo da mesma. Note que este primeiro prato a ser retirado foi o último a ser colocado na pilha e que o primeiro prato a ser colocado na pilha será o último a ser retirado.

De forma análoga ao nosso exemplo da pilha de pratos, a pilha de execução do nosso algoritmo se inicia como *Procedimento1* e continua como indicado a seguir:

1º momento:	**2º momento:** Procedimento1	**3º momento:** Procedimento2 Procedimento1	**4º momento:** Procedimento3 Procedimento2 Procedimento1
5º momento: Procedimento2 Procedimento1	**6º momento:** Procedimento1	**7º momento:**	

Note que o *Procedimento1* foi o primeiro a ser inserido na pilha. Esse procedimento faz uma chamada ao *Procedimento2*. Ao ser chamado, o *Procedimento2* é inserido na pilha e torna-se o método que está sendo executado. Quando o *Procedimento2* faz uma chamada ao *Procedimento3*, este último entra na pilha. Ao terminar a sua execução, o *Procedimento3* é desempilhado, o que faz com que o *Procedimento2* se torne o procedimento em processamento, sendo que o *Procedimento2* estava anteriormente parado na linha em que fazia a chamada ao *Procedimento3*, aguardando que este último terminasse para que ele pudesse continuar. Ao término do *Procedimento2*, o mesmo deixa a pilha e o *Procedimento1* prossegue a sua execução.

Vamos agora trabalhar com um procedimento recursivo, cujo objetivo é imprimir uma mensagem passada por parâmetro um número *n* de vezes, sendo *n* também passado por parâmetro sob o nome "numeroVezes".

```
static void Imprimir(string mensagem, int numeroVezes)
{
    if (numeroVezes > 0)
    {
        Console.WriteLine($"{numeroVezes}) {mensagem}");
        Imprimir(mensagem, numeroVezes - 1);
    }
}

static void Main(string[] args)
{
    Imprimir("Teste de mensagem", 5);
    Console.ReadKey();
}
```

Veja a saída do algoritmo:

```
5) Teste de mensagem
4) Teste de mensagem
3) Teste de mensagem
2) Teste de mensagem
1) Teste de mensagem
```

Agora vamos fazer uma pequena alteração no algoritmo que nos permitirá visualizar na saída o que está acontecendo na pilha de execução do mesmo. Segue abaixo a alteração no algoritmo e a saída obtida após a sua execução.

```
static void Imprimir(string mensagem, int numeroVezes)
{
    Console.WriteLine($"Empilhou. Valor parâmetro: {numeroVezes}");

    if (numeroVezes > 0)
    {
        Console.WriteLine($"  {numeroVezes}) {mensagem}");
        Imprimir(mensagem, numeroVezes - 1);
    }

    Console.WriteLine($"Desempilhou. Valor parâmetro: {numeroVezes}");
}
```

```
Empilhou. Valor parâmetro: 5
  5) Teste de mensagem
Empilhou. Valor parâmetro: 4
  4) Teste de mensagem
Empilhou. Valor parâmetro: 3
  3) Teste de mensagem
Empilhou. Valor parâmetro: 2
  2) Teste de mensagem
Empilhou. Valor parâmetro: 1
  1) Teste de mensagem
Empilhou. Valor parâmetro: 0
Desempilhou. Valor parâmetro: 0
Desempilhou. Valor parâmetro: 1
Desempilhou. Valor parâmetro: 2
Desempilhou. Valor parâmetro: 3
Desempilhou. Valor parâmetro: 4
Desempilhou. Valor parâmetro: 5
```

Note que a cada chamada do procedimento *Imprimir* é apresentada na tela a mensagem e também é realiza uma nova chamada ao próprio procedimento *Imprimir*. Quando o valor do segundo parâmetro passado é zero, a chamada recursiva não é feita, o que acaba permitindo ao procedimento que está no topo da pilha terminar o seu processamento. Com o término de um procedimento, o próximo procedimento da pilha pode terminar o seu processamento e assim sucessivamente até o término da execução do programa.

Como os números antes dos parênteses de cada mensagem foram impressos antes da chamada recursiva, os valores ficaram em ordem decrescente. Veja a seguir uma alteração no algoritmo que permite que esses números sejam mostrados em ordem crescente e a respectiva saída do algoritmo após essa alteração.

```
static void Imprimir(string mensagem, int numeroVezes)
{
    Console.WriteLine($"Empilhou. Valor parâmetro: {numeroVezes}");

    if (numeroVezes > 0)
    {
        Imprimir(mensagem, numeroVezes - 1);
        Console.WriteLine($"  {numeroVezes}) {mensagem}");
    }

    Console.WriteLine($"Desempilhou. Valor parâmetro: {numeroVezes}");
}
```

```
Empilhou. Valor parâmetro: 5
Empilhou. Valor parâmetro: 4
Empilhou. Valor parâmetro: 3
Empilhou. Valor parâmetro: 2
Empilhou. Valor parâmetro: 1
Empilhou. Valor parâmetro: 0
Desempilhou. Valor parâmetro: 0
  1) Teste de mensagem
Desempilhou. Valor parâmetro: 1
  2) Teste de mensagem
Desempilhou. Valor parâmetro: 2
  3) Teste de mensagem
Desempilhou. Valor parâmetro: 3
  4) Teste de mensagem
Desempilhou. Valor parâmetro: 4
  5) Teste de mensagem
Desempilhou. Valor parâmetro: 5
```

Como a chamada recursiva foi feita antes da impressão da mensagem, somente após a sexta chamada (feita com o valor do parâmetro igual a zero) ser concluída, é que a primeira impressão é executada e esta é justamente a impressão cujo comando se encontra no quinto procedimento a ser empilhado, sendo que neste o valor do parâmetro é igual a 1.

TÓPICO IV – SUBSTITUIÇÃO DE ITERAÇÃO POR RECORRÊNCIA

Um algoritmo iterativo pode ser substituído por uma versão recursiva seguindo os seguintes passos:

1. Substitua o comando *for* ou o comando *do..while* pelo comando *while*.
2. Altere a sub-rotina para passar a receber por parâmetro o valor a ser processado.
3. Substitua o comando de atualização da variável de controle do comando *while* pela chamada recursiva passando por parâmetro o próximo valor dessa variável.
4. Substitua o comando *while* por um comando *if*.
5. Crie uma sub-rotina empacotadora para fazer a primeira chamada à sub-rotina recursiva.

Veja abaixo um exemplo de algoritmo iterativo para cálculo dos divisores de um número natural.

```
static void ImprimirDivisores(int numero)
{
    for (int divisor = 1; divisor <= numero; divisor++)
    {
        if (numero % divisor == 0)
            Console.Write($"{divisor} ");
    }
}
```

Seguindo os cinco passos para transformar esse procedimento em um procedimento recursivo, temos:

1. Substitua o comando for ou o comando *do..while* pelo comando *while*.

```
static void ImprimirDivisores(int numero)
{
    int divisor = 1;

    while (divisor <= numero)
    {
        if (numero % divisor == 0)
            Console.Write($"{divisor} ");

        divisor++;
    }
}
```

2. Altere a sub-rotina para passar a receber por parâmetro o valor a ser processado.

```
static void ImprimirDivisores(int numero, int divisor)
{
    while (divisor <= numero)
    {
        if (numero % divisor == 0)
            Console.Write($"{divisor} ");

        divisor++;
    }
}
```

3. Substitua o comando de atualização da variável de controle do comando *while* pela chamada recursiva passando por parâmetro o próximo valor dessa variável.

```
static void ImprimirDivisores(int numero, int divisor)
{
    while (divisor <= numero)
    {
        if (numero % divisor == 0)
            Console.Write($"{divisor} ");

        ImprimirDivisores(numero, divisor + 1);
    }
}
```

4. Substitua o comando *while* por um comando *if*.

```
static void ImprimirDivisores(int numero, int divisor)
{
    if (divisor <= numero)
    {
        if (numero % divisor == 0)
            Console.Write($"{divisor} ");

        ImprimirDivisores(numero, divisor + 1);
    }
}
```

5. Crie uma função empacotadora para fazer a primeira chamada à sub-rotina recursiva.

```
static void ImprimirDivisores(int numero)
{
    ImprimirDivisores(numero, 1);
}
```

```
static void ImprimirDivisores(int numero, int divisor)
{
    if (divisor <= numero)
    {
        if (numero % divisor == 0)
            Console.Write($"{divisor} ");

        ImprimirDivisores(numero, divisor + 1);
    }
}
```

Se você não quiser criar a sub-rotina empacotadora, em C# você pode definir o valor padrão para um argumento de uma sub-rotina se o mesmo não for passado. Adicionando o valor 1 como padrão para o argumento divisor da nossa função, ao não passarmos esse parâmetro, será assumido o valor 1 para o mesmo. Dessa forma, podemos chamar a função passando apenas o valor para o parâmetro chamado *numero* e deixando o parâmetro chamado divisor ser passado somente nas chamadas recursivas. Veja o único procedimento abaixo.

```
static void ImprimirDivisores(int numero, int divisor = 1)
{
    if (divisor <= numero)
    {
        if (numero % divisor == 0)
            Console.Write($"{divisor} ");

        ImprimirDivisores(numero, divisor + 1);
    }
}
```

Agora teste esse procedimento com a seguinte chamada:

```
ImprimirDivisores(150);
```

RESUMO DA UNIDADE I - ATIVIDADES

Uma sub-rotina não recursiva é aquela que possui somente chamadas externas. Já uma sub-rotina recursiva é aquela que possui ao menos uma chamada interna, sendo esta direta ou indireta (GOODRICH, TAMASSIA, 2011).

A recursividade em procedimentos ou funções pode ser direta, caso em que a sub-rotina chama a si mesmo diretamente, ou pode ser indireta, caso em que uma sub-rotina B chamada pela sub-rotina A realiza uma chamada à sub-rotina A (CELES, CERQUEIRA, RANGEL, 2017).

Em uma pilha de recursão, as sub-rotinas são retiradas na pilha na ordem inversa à que foram inseridas, ou seja, a primeira a ser chamada é a última a ser removida.

As implementações iterativas geralmente apresentam melhor desempenho, porém alguns algoritmos recursivos podem ser mais concisos que as suas versões recursivas (SZWARCFITER, MARKEZON, 2010).

Exercícios

1) Qual o resultado da execução do trecho de código abaixo?

```
static void ImprimirTexto(string texto)
{
    Console.WriteLine(texto);
    ImprimirTexto(texto);
}
```

   a) O valor da variável *texto* ficará sendo impresso repetidas vezes sem parar. A pilha de recursão pode ocupar toda a memória do computador se o sistema operacional ou o usuário não intervir.
   b) O programa ficará travado sem apresentar nada ao usuário. A pilha de recursão pode ocupar toda a memória do computador se o sistema operacional ou o usuário não intervir.

2) Qual o resultado da execução do trecho de código abaixo?

```
static void ImprimirTexto(string texto)
{
    ImprimirTexto(texto);
    Console.WriteLine(texto);
}
```

   a) O valor da variável *texto* ficará sendo impresso repetidas vezes sem parar. A pilha de recursão pode ocupar toda a memória do computador se o sistema operacional ou o usuário não intervir.
   b) O programa ficará travado sem apresentar nada ao usuário. A pilha de recursão pode ocupar toda a memória do computador se o sistema operacional ou o usuário não intervir.

3) Veja o trecho de código abaixo:

```
static void A(int n)
{
    if (n > 0)
        A(n - 1);
}
```

Sobre esse trecho de código, assinale a alternativa correta:

a) Apresenta um exemplo de recursividade direta.
b) Apresenta um exemplo de recursividade indireta.
c) Não apresenta recursividade.
d) É um exemplo de cálculo do fatorial de um número.
e) Possui iteração ao invés de recorrência.

4) Veja o trecho de código abaixo:

```
static void B(int n)
{
    if (n > 0)
        C(n);
}

static void C(int n)
{
    B(n - 1);
}
```

Sobre esse trecho de código, assinale a alternativa correta:

a) Apresenta um exemplo de recursividade direta.
b) Apresenta um exemplo de recursividade indireta.
c) Não apresenta recursividade.
d) É um exemplo de cálculo do fatorial de um número.
e) Possui iteração ao invés de recorrência.

5) A seguir se encontram duas sub-rotinas, sendo a primeira uma sub-rotina empacotadora da segunda sub-rotina, sendo esta última recursiva. Assinale a alternativa que indique a finalidade dessas sub-rotinas.

```
static bool X(int n)
{
    if (n < 1)
        throw new Exception("Número inválido!");

    return X(n, n / 2);
}
```

```
static bool X(int n, int d)
{
    if (d > 1)
        if (n % d != 0)
            return X(n, d - 1);

    return d == 1;
}
```

a) Retornar se o número n pode ser dividido por um número ímpar.
b) Calcular o mínimo divisor comum de um número n.
c) Checar se o número n é par ou ímpar.
d) Calcular o fatorial do número n.
e) Retornar se o número n é um número primo.

6) Faça um procedimento recursivo que receba por parâmetro um vetor vet de números reais e, multiplique por -1 todos os elementos negativos desse vetor. Para esse exercício não se pode utilizar as estruturas de repetição (*for*, *while* e *do while*).

7) Faça uma função recursiva que receba um número inteiro n por parâmetro e retorne a soma dos números inteiros entre zero e n. A função deve funcionar adequadamente tanto para n positivo quanto para negativo. Para esse exercício não se pode utilizar as estruturas de repetição (*for*, *while* e *do while*).

Proposta de Gabarito

1) A
2) B
3) A
4) B
5) E
6)

```
private static void TornaPositivo(double[] vet, int indice = 0)
{
    if (indice < vet.Length)
    {
        if (vet[indice] < 0)
            vet[indice] *= -1;

        TornaPositivo(vet, indice + 1);
    }
}
```

7)

```
private static int SomaAteZero(int n)
{
    if (n > 0)
        return n + SomaAteZero(n - 1);
    else if (n < 0)
        return n + SomaAteZero(n + 1);
    else
        return 0;
}
```

UNIDADE II – ORDENAÇÃO

Uma das tarefas recorrentes em programação é a necessidade de ordenação de valores dentro de um vetor ou outra estrutura para armazenamento de dados. É bastante comum encontrar nas bibliotecas ou no framework de sua linguagem de programação métodos de ordenação que podem ser estendidos para ordenar um conjunto de objetos qualquer.

Ainda que existam vários métodos já implementados, é muito importante entendermos diferentes métodos de ordenação, pois esses métodos constituem um excelente conjunto de algoritmos que são importantes para aprimorar a nossa lógica de programação.

O processo de rearranjar os elementos, em ordem ascendente ou descendente, de um conjunto de objetos é conhecido como ordenação.

A ordenação pode ser utilizada para facilitar a recuperação de dados e o seu uso é comum e muito importante quando pensamos em otimização de desempenho computacional (CELES, CERQUEIRA, RANGEL, 2017).

Os métodos de ordenação são comumente baseados em comparação de elementos, embora outras abordagens também sejam possíveis.

Vamos separar os métodos de ordenação em dois grupos: métodos simples e métodos eficientes. Para esses dois métodos, considere N como sendo o número de elementos do conjunto a ser ordenado.

Os métodos simples são mais adequados para pequenos conjuntos e sua complexidade de comparação é de $O(N^2)$, ou seja, cresce em acordo com o quadrado do tamanho do conjunto.

Os métodos eficientes geralmente utilizam comparações mais complexas, porém em menor quantidade. Esses métodos são mais adequados para grandes conjuntos, pois a sua complexidade de comparação é de O(n log n).

Nessa disciplina, analisaremos os métodos simples:

- Método bolha (*bubblesort*)
- Método da inserção direta (*insertionsort*)
- Método da seleção direta (*selectionsort*)

O único método eficiente que será analisado nessa disciplina é o *quicksort*.

Como estudo complementar, você pode buscar mais informações sobre o método da intercalação simples (*shellsort*) e sobre o método da seleção em árvores (*heapsort*). O primeiro método é um método simples e o segundo é eficiente.

Nessa apostila, em alguns algoritmos será chamado um procedimento auxiliar para trocar valores no vetor. O código desse procedimento se encontra abaixo.

```
static void Troca(double[] vetor, int i, int j)
{
    //Guarda o valor armazenado na posição i do vetor.
    double aux = vetor[i];
    vetor[i] = vetor[j];
    vetor[j] = aux;
}
```

Nesse ponto, já podemos explorar algum método de ordenação. Faremos isso no próximo tópico.

TÓPICO I – O MÉTODO BOLHA (*BUBBLESORT*)

Na ordenação por trocas, o processo de ordenação se dá pela comparação de pares de chaves de ordenação, sendo trocados elementos que estiverem fora de ordem.

O método bolha é um método de ordenação por trocas em que o vetor é percorrido comparando, se os elementos da posição i e da posição $i + 1$ estão ordenados. Os elementos são trocados sempre que forem encontrados pares desordenados. Esse processo é repetido até o vetor seja percorrido sem que nenhuma troca seja feita (CELES, CERQUEIRA, RANGEL, 2017).

A função que você utilizará para indicar qual a ordenação necessária pode variar, sendo mais comum a ordenação de números do menor para o maior ou do maior para o menor ou, no caso de informação textual, a ordenação lexicográfica.

Agora já podemos implementar um algoritmo para o método bolha, conforme pode ser visto a seguir.

```
static void BubbleSort(double[] vetor)
{
    int fim = vetor.Length - 1, pos = 0;
    bool troca = true;
    while (troca)
    {
        troca = false;
        for (int i = 0; i < fim; i++) //varredura i
            if (vetor[i] > vetor[i + 1])
            {
                Troca(vetor, i, i + 1);
                pos = i;
                troca = true;
            }
        fim = pos – 1;
    }
}
```

No algoritmo acima, a variável troca é inicializada com o valor *true*, de forma que o primeiro teste do comando de repetição passe e a iteração se inicie. A variável troca então é passada para *false* e só receberá o valor *true* novamente se, no laço mais interno, houver ao menor um par de elementos não ordenados.

O laço mais interno do algoritmo é utilizado para variar o valor da variável i entre 0, que é o índice da primeira posição do vetor, e a penúltima posição do vetor. Para cada elemento *i*, é testado se esse elemento é menor do que o da posição *i* + 1, caso em que os elementos serão trocados. Note que essa condição nos permite afirmar que os elementos do vetor no algoritmo acima estarão ordenados em ordem crescente após a execução do procedimento *BubbleSort*.

Para ordenar um vetor de números reais, o código abaixo poderia ser executado.

```
static void Main(string[] args)
{
    var vet = new double[] { 1.4, 2.3, 0, 7.8, 3.1 };

    BubbleSort(vet);

    Console.ReadKey();
}
```

Quanto à análise do método bolha, temos o melhor caso quando o vetor está ordenado. Nesse caso, a primeira varredura será feita e, como nenhum elemento estará desordenado, o procedimento de ordenação será finalizado. Esse caso necessita de n – 1 comparações.

Já o pior caso é o que os elementos estão inversamente ordenados. Para esse caso, um elemento do vetor será posicionado em seu local definitivo a cada varredura. O total de comparações para esse caso será a soma da seguinte progressão aritmética:

$$(n-1)+(n-2)+\cdots+2+1 = \frac{(n^2-n)}{2}$$

TÓPICO II – O MÉTODO DA INSERÇÃO (*INSERTIONSORT*)

Esse método consiste em inserir cada elemento em sua posição correta dentre os elementos já ordenados. O processo é semelhante à ordenação de um baralho, onde inicialmente dividimos os elementos em dois subconjuntos, sendo o primeiro subconjunto o dos elementos ordenados, que contém inicialmente somente um elemento. O segundo subconjunto é o dos elementos não ordenados. A cada iteração, pega-se um elemento do subconjunto desordenado e o insere na posição correta do subconjunto ordenado.

Se estivermos utilizando um vetor, podemos dividi-lo em dois segmentos, sendo um o dos elementos ordenados e o outro os dos elementos não ordenados.

```
static void Insercao(double[] vetor)
{
    //a parte ordenada está entre 0 e i
    for (int i = 1; i < vetor.Length; i++)
    {
        //valor a ser inserido na parte ordenada
        double chave = vetor[i];
        int j = i - 1;
        //localiza a posição j de Chave na parte ordenada
        while (j >= 0 && chave < vetor[j])
        {
            vetor[j + 1] = vetor[j];
            j--;
        }
        vetor[j + 1] = chave;
    }
}
```

No laço externo, a variável *i* é inicializada com o valor 1, pois este corresponde ao índice do segundo elemento do vetor. Com isso, estamos dividindo o nosso vetor em dois segmentos, sendo o que se inicia e termina no índice zero o segmento dos elementos ordenados e o que se inicia em 1 e termina no índice do último elemento do vetor, o segmento dos elementos não ordenados.

A variável *j* é inicializada com o valor do elemento anterior ao elemento *i*. Com isso, o laço interno irá deslocar para a direita todos os elementos maiores que os que se encontram na posição *i*. Ao término do laço interno, o elemento *i* é inserido na posição *j* + 1.

Para ordenar um vetor de números reais, o código abaixo poderia ser executado.

```
static void Main(string[] args)
{
    var vet = new double[] { 1.4, 2.3, 0, 7.8, 3.1 };

    Insercao(vet);

    Console.ReadKey();
}
```

Analisando o método da inserção, temos que o melhor caso é aquele em que o vetor já se encontra ordenado, onde será necessária ao menos uma comparação para localizar a posição da chave. Para esse caso, o método efetuará um total de $n - 1$ iterações até que o vetor seja considerado ordenado.

O pior caso para o método da inserção é aquele em que o vetor se encontra inversamente ordenado, pois cada elemento a ser inserido será menor que todos os anteriormente ordenados, fazendo com que todos os elementos já ordenados sejam deslocados uma posição à direita a cada iteração. O total de comparações para esse caso será a soma da seguinte progressão aritmética:

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{(n^2 - n)}{2}$$

TÓPICO III – O MÉTODO DA SELEÇÃO (*SELECTIONSORT*)

Nesse método, para uma ordenação crescente, o menor elemento será selecionado a cada iteração e o mesmo será trocado com primeiro elemento do vetor. Em seguida, o segundo menor elemento será trocado com o segundo elemento do vetor e assim sucessivamente, até que o vetor esteja ordenado.

Para uma ordenação descendente, basta modificar o método exposto acima de forma que seja selecionado o maior elemento ao invés do menor elemento.

Com o método da seleção, também teremos dois segmentos, pois ao se adicionar um elemento em sua posição correta, o início do segmento dos valores não ordenados é deslocado para a direita.

No início, o segmento dos elementos ordenados é vazio e o segmento dos valores não ordenados possui todos os outros elementos. Em seguida, é feita uma varredura no segmento que possui os elementos não selecionados para encontrar o menor (ou maior) elemento do vetor. O elemento encontrado é inserido na última posição do segmento ordenado, sendo que esse último passo pode ser feito trocando o elemento selecionado pelo elemento que se encontra à direita do último elemento do segmento dos elementos ordenados.

```
static void Selecao(double[] vetor)
{
    //a parte ordenada está entre 0 (inclusive) e i (exclusive)
    for (int i = 0; i < vetor.Length - 1; i++)
    {
        int indMenor = i;
        //localiza o índice IndMenor do menor elemento
        //na parte não ordenada do vetor
        for (int j = i + 1; j < vetor.Length; j++)
        {
            if (vetor[j] < vetor[indMenor])
                indMenor = j;
        }
        if (i != indMenor)//troca o item i pelo item indMenor
            Troca(vetor, i, indMenor);
    }
}
```

No algoritmo acima, a variável indMenor é sempre inicializada com o valor da variável i. Os elementos já ordenados são aqueles que se encontram antes da posição i. O laço interno procura à partir da posição seguinte à i o menor elemento. Ao término do laço interno, se o menor elemento encontrado não estiver na posição i, ele é trocado pelo elemento da posição i. Na próxima iteração, a variável i será incrementada, o que fará com que o segmento dos valores ordenados tenha o seu tamanho incrementado em 1, pois a parte ordenada fica entre os índices zero e a posição anterior à i. Já a parte não ordenada se encontra entre os elementos i e o último elemento do vetor.

O total de comparações para esse caso será a soma da seguinte progressão aritmética:

$$(n-1)+(n-2)+\cdots+2+1 \;=\; \frac{(n^2-n)}{2}$$

Para ordenar um vetor de números reais, o código abaixo poderia ser executado.

```
static void Main(string[] args)
{
    var vet = new double[] { 1.4, 2.3, 0, 7.8, 3.1 };

    Selecao(vet);

    Console.ReadKey();
}
```

O desempenho médio do método é $O(n^2)$, ou seja, é proporcional ao quadrado do número de elementos do vetor.

No quadro a seguir, temos a análise da complexidade de cada método de ordenação visto até o momento.

	Números de Comparações		**Números de Trocas**	
	Melhor	Pior	Melhor	Pior
Ord. por Seleção	$O(n^2)$	$O(n^2)$	O(n)	O(n)
Ord. por Bolhas	O(n)	$O(n^2)$	O(1)	$O(n^2)$
Ord. por Inserção	O(n)	$O(n^2)$	O(n)	$O(n^2)$

TÓPICO IV – QUICKSORT

O quicksort é um método eficiente de ordenação. Ele é amplamente utilizado devido ao seu bom desempenho para a ordenação de grandes conjuntos de dados somada à sua facilidade de implementação.

O quicksort se inicia considerando a partição contendo todos os elementos do vetor. Em seguida, um elemento desse vetor é escolhido como pivô. Nesse ponto o algoritmo colocará à esquerda do pivô todos os elementos menores que o próprio pivô. Os elementos maiores ficam à direita do pivô. Nesse ponto, o pivô já estará em sua posição correta. O passo seguinte é chamar o quicksort para os elementos que estão à esquerda do pivô e também chamar o quicksort para os elementos que estão à direita do pivô.

No método quicksort é comum utilizar uma variável para indicar qual o índice de um dos elementos maiores que o pivô e qual o índice de um dos elementos menores que o pivô. Para isso, basta colocar o índice do maior elemento no início da partição e o índice do menor elemento ao final. A estratégia é caminhar com o índice do elemento maior que o pivô para o final da partição enquanto o valor do elemento que este índice aponta for menor ou igual ao pivô. Também se deve caminhar com o índice do elemento menor que o pivô no sentido do início do vetor enquanto o elemento dessa posição for maior que o pivô. O objetivo disso é encontrar um elemento maior que o pivô que se encontra à esquerda dele e encontrar um elemento menor que o pivô que se encontra à direita dele. Encontrados os elementos desposicionados, os mesmos são trocados (GOODRICH, TAMASSIA, 2011).

Abaixo é apresentado um procedimento que ordena um vetor de números inteiros utilizando para isso o método quicksort.

```
static void Quicksort(double[] vetor)
{
    Quicksort(vetor, 0, vetor.Length - 1);
}

static void Quicksort(double[] vetor, int inicio, int fim)
{
    if (inicio >= fim)
        return;

    int indiceMenor = fim, indiceMaior = inicio;

    var pivo = indiceMenor;

    do
    {
        while (indiceMaior < fim
        && vetor[indiceMaior] <= vetor[pivo])
            indiceMaior++;
        while (indiceMenor > inicio
        && vetor[indiceMenor] > vetor[pivo])
            indiceMenor--;
        if (indiceMaior < indiceMenor)
            Troca(vetor, indiceMaior, indiceMenor);
    } while (indiceMenor > indiceMaior);

    Troca(vetor, indiceMenor, pivo);

    Quicksort(vetor, inicio, pivo - 1);
    Quicksort(vetor, pivo + 1, fim);
}
```

Para ordenar um vetor de números reais, o código abaixo poderia ser executado.

```
static void Main(string[] args)
{
    var vet = new double[] { 1.4, 2.3, 0, 7.8, 3.1 };

    Quicksort(vet);

    Console.ReadKey();
}
```

Quanto à análise do quicksort, no melhor caso, a partição divide o vetor em dois sub-vetores de mesmo tamanho e o procedimento quicksort é chamado para cada sub-vetor. A profundidade da árvore de recursão será O(n log n) e o número de acessos na partição é O(n). O tempo de execução para o pior caso é de $O(n \log_2 n)$. Já no pior caso, o vetor será particionado em um sub-vetor de tamanho zero e um outro de tamanho igual ao tamanho do sub-vetor menos 1. Para esse caso, a profundidade da árvore de recursão será de O(n) e o número de acessos por partição O(n), o que resulta em um tempo de execução de $O(n^2)$ (GOODRICH, TAMASSIA, 2011).

RESUMO DA UNIDADE II - ATIVIDADES

O processo de rearranjar os elementos, em ordem ascendente ou descendente, de um conjunto de objetos é conhecido como ordenação.

A ordenação pode ser utilizada para facilitar a recuperação de dados e o seu uso é comum e muito importante quando pensamos em otimização de desempenho computacional.

O método bolha é um método de ordenação por trocas em que o vetor é percorrido comparando, se os elementos da posição i e da posição i + 1 estão ordenados. Os elementos são trocados sempre que forem encontrados pares desordenados. Esse processo é repetido até o vetor seja percorrido sem que nenhuma troca seja feita.

O método da inserção consiste em inserir cada elemento em sua posição correta dentre os elementos já ordenados. O processo é semelhante à ordenação de um baralho, onde inicialmente dividimos os elementos em dois subconjuntos, sendo o primeiro subconjunto o dos elementos ordenados, que contém inicialmente somente um elemento. O segundo subconjunto é o dos elementos não ordenados. A cada iteração, pega-se um elemento do subconjunto desordenado e o insere na posição correta do subconjunto ordenado.

No método da seleção, o menor elemento será selecionado a cada iteração e o mesmo será trocado com primeiro elemento do vetor. Em seguida, o segundo menor elemento

será trocado com o segundo elemento do vetor e assim sucessivamente, até que o vetor esteja ordenado.

O quicksort é um método eficiente de ordenação. Ele é amplamente utilizado devido ao seu bom desempenho para a ordenação de grandes conjuntos de dados somada à sua facilidade de implementação.

O quicksort se inicia considerando a partição contendo todos os elementos do vetor. Em seguida, um elemento desse vetor é escolhido como pivô. Nesse ponto o algoritmo colocará à esquerda do pivô todos os elementos menores que o próprio pivô. Os elementos maiores ficam à direita do pivô. Nesse ponto, o pivô já estará em sua posição correta. O passo seguinte é chamar o quicksort para os elementos que estão à esquerda do pivô e também chamar o quicksort para os elementos que estão à direita do pivô.

Exercícios

1) Observe o algoritmo abaixo que manipula um vetor de seis posições preenchido com os elementos {50, 60, 30, 40, 10, 20}. Qual será conteúdo do vetor após a execução do algoritmo?

```
static void X(double[] vetor)
{
    int fim = vetor.Length - 1, pos = 0;
    bool troca = true;
    while (troca)
    {
        troca = false;
        for (int i = 0; i < fim; i++)
            if (vetor[i] < vetor[i + 1])
            {
                Troca(vetor, i, i + 1);
                pos = i;
                troca = true;
            }
        fim = pos – 1;
    }
}

static void Troca(double[] vetor, int i, int j)
{
    var aux = vetor[i];
    vetor[i] = vetor[j];
    vetor[j] = aux;
}
```

a) 60, 50, 40, 30, 20, 10
b) 10, 20, 30, 40, 50, 60
c) 40, 50, 60, 10, 20, 30
d) 40, 50, 60, 40, 50, 60

2) Observe o algoritmo abaixo que manipula um vetor de seis posições preenchido com os elementos {5, 6, 3, 4, 1, 2}. Qual será conteúdo do vetor após a execução do algoritmo?

```
static void Y(double[] vetor)
{
    for (int i = 1; i < vetor.Length; i++)
    {
        var chave = vetor[i];
        var j = i - 1;
        while (j >= 0 && chave < vetor[j])
        {
            vetor[j + 1] = vetor[j];
            j--;
        }
        vetor[j + 1] = chave;
    }
}

static void Troca(double[] vetor, int i, int j)
{
    var aux = vetor[i];
    vetor[i] = vetor[j];
    vetor[j] = aux;
}
```

a) 6, 5, 4, 3, 2, 1
b) 1, 2, 3, 4, 5, 6
c) 4, 5, 6, 1, 2, 3
d) 4, 5, 6, 4, 5, 6

3) Qual o método de ordenação que, após a escolha do pivô, transfere todos os elementos menores que o pivô para a esquerda do elemento pivô e também transfere todos os elementos maiores que o pivô para a direita do elemento pivô?

a) Insertionsort
b) Selectionsort
c) Bubblesort
d) Quicksort

4) Dentre os métodos de ordenação estudado, qual é o que apresenta o melhor desempenho para ordenação de vetores grandes com elementos randomicamente distribuídos?

   a) Insertionsort
   b) Selectionsort
   c) Bubblesort
   d) Quicksort

5) Qual o método de ordenação que se assemelha à ordenação de cartas em um baralho?

   a) Insertionsort
   b) Selectionsort
   c) Bubblesort
   d) Quicksort

6) Considere a classe Produto abaixo e faça um procedimento que receba por parâmetro um vetor de produtos e ordene esse vetor pelo código do produto. Utilize o método da inserção para ordenar o vetor.

```
class Produto
{
    public int Codigo { get => codigo; set => codigo = value; }
    public string Descricao
    {
        get => descricao;
        set => descricao = value;
    }
    public double Preco { get => preco; set => preco = value; }

    private int codigo;
    private string descricao;
    private double preco;
}
```

7) Considere a classe Produto da questão 6 e faça um procedimento que receba por parâmetro um vetor de produtos e ordene esse vetor pelo preço do produto. Utilize o método da seleção para ordenar o vetor.

Proposta de Gabarito

1) A
2) B
3) D
4) D
5) A
6)

```
static void Insercao(Produto[] vetor)
{
    for (int i = 1; i < vetor.Length; i++)
    {
        var chave = vetor[i];
        var j = i - 1;
        while (j >= 0 && chave.Codigo < vetor[j].Codigo)
        {
            vetor[j + 1] = vetor[j];
            j--;
        }
        vetor[j + 1] = chave;
    }
}
```

7)

```
static void Selecao(Produto[] vetor)
{
    for (int i = 0; i < vetor.Length - 1; i++)
    {
        var indMenor = i;
        for (int j = i + 1; j < vetor.Length; j++)
        {
            if (vetor[j].Preco < vetor[indMenor].Preco)
                indMenor = j;
        }
        if (i != indMenor)
            Troca(vetor, i, indMenor);
    }
}
```

UNIDADE II – LISTAS LINEARES

Listas lineares são utilizadas em diversas tarefas computacionais. Como vetores são estruturas estáticas e não podem ser expandidas, o seu uso em problemas que trabalham com quantidades variáveis de dados pode não ser adequado. Você pode encontrar diferentes tipos de listas lineares nas bibliotecas ou no framework de sua linguagem de programação, porém o estudo detalhado dessas estruturas é importante para o entendimento das diferentes listas e suas aplicações. Esse conteúdo também o ajudará a aprimorar a sua lógica de programação.

Uma lista é uma estrutura linear, composta por um conjunto de elementos chamados nós. Os nós são organizados de forma a manter uma relação entre os mesmos. Considere os n elementos $x_0, x_1, x_2, \ldots, x_{n-1}$ de uma lista linear. Nesse caso:

Para $n > 0$:

x_0 é o primeiro nó

x_{n-1} é o último nó

$\forall k \in \mathbb{Z}$ e $0 < k < n$-1, o nó x_{k-1} precede o nó x_k, que por sua vez é sucedido pelo nó x_{k+1}.

Uma lista linear é uma estrutura dinâmica cujo comprimento da lista, que é o seu número de nós, pode ser alterado em tempo de execução (CELES, CERQUEIRA, RANGEL, 2017).

Algumas operações comuns em listas lineares são: consultar um determinado nó, atribuir um novo valor a um nó, excluir um nó e inserir um novo nó na lista. Um nó pode ser inserido na lista em diferentes posições, como no início da lista, no final da lista ou antes ou depois de um determinado elemento (SWARCFITER, MARKENZON, 1994).

Existem diferentes formas de representar uma lista linear, sendo as mais comuns a contiguidade de nós e a encadeamento de nós.

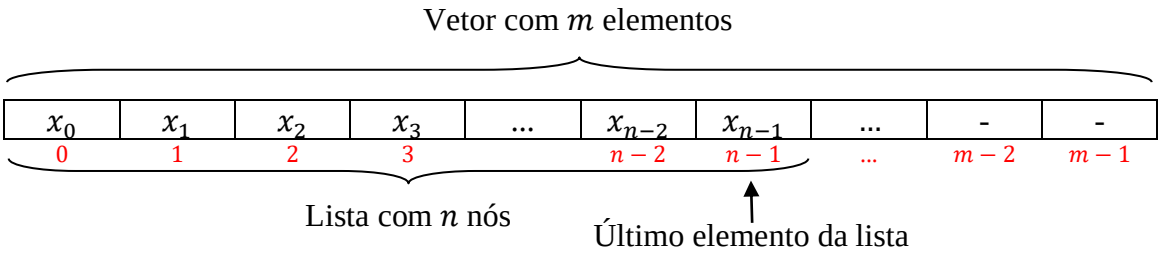
Nesse ponto, já podemos iniciar o estudo de uma dessas listas. Faremos isso no próximo tópico.

TÓPICO I – LISTAS CONTÍGUAS

Listas contíguas são listas em que os seus elementos são alocados fisicamente em posições consecutivas da memória. Devido ao armazenamento consecutivo, é comum a utilização de vetores para armazenar os elementos de uma lista contígua (SWARCFITER, MARKENZON, 1994).

Considerando que uma lista deve ter tamanho variável e que um vetor tem tamanho estático, é possível identificar que o tamanho do vetor utilizado limita o tamanho da lista. Dessa forma, o vetor utilizado tende a ter um tamanho que é considerado suficiente para o problema que se deseja resolver. Já o tamanho da lista deve ser controlado por variáveis que identificam ao menos qual o último nó da lista.

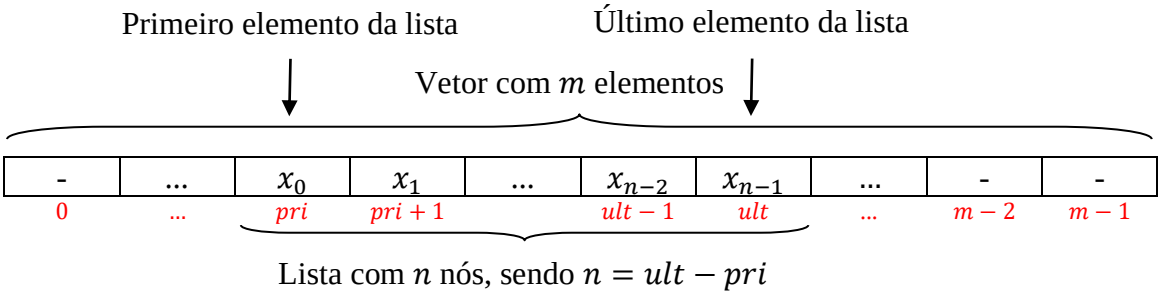
Abaixo é representada uma lista contígua com n elementos que utiliza um vetor com m elementos para o seu armazenamento.



Nessa representação, o último elemento da lista poderia ser representado por uma variável. Uma opção à variável que representa o último elemento da lista seria uma variável que representa o tamanho da lista. Como o primeiro elemento da nossa lista recebe o índice zero, o tamanho n da lista pode ser representado em uma variável e o último elemento da lista poderia ser obtido no índice anterior a n, ou seja, no índice $n - 1$.

Note que para inserir elementos ao final da lista, bastaria inseri-lo na posição n e incrementar em uma unidade o tamanho n da lista (n++). Já para inserir um elemento no início da lista, é necessário deslocar cada elemento da lista uma posição à frente e inserir o elemento na primeira posição. Dessa forma, o processo de inserir um elemento no início dessa lista não apresentará um bom desempenho, pois o seu processamento crescerá em acordo com o tamanho do vetor.

Outra representação de uma lista contígua segue abaixo.



Note que agora é necessária a utilização de duas variáveis, sendo a variável pri para representar o início da lista e a variável ult para representar o final da lista. Já o tamanho n da lista pode ser calculado da seguinte forma: $n = ult - pri$.

Para adicionar um elemento ao final dessa lista, basta adicioná-lo na posição $ult + 1$ e incrementar em uma unidade o valor da variável ult.

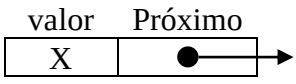
A vantagem dessa segunda implementação sobre a primeira é que na segunda representação temos uma maior facilidade para adicionar elementos no início da lista. Para isso, basta adicionar o elemento na posição $pri - 1$ e decrementar em uma unidade o valor da variável pri (pri--).

TÓPICO II – LISTA SIMPLESMENTE ENCADEADA

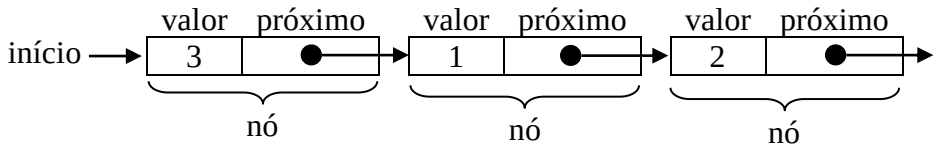
A lista simplesmente encadeada, assim como as listas apresentadas nos próximos tópicos, são listas dinâmicas que não armazenam os seus nós em posições sequenciais na memória. Essas listas não possuem o limite de elementos imposto pela estrutura estática utilizada nas listas contíguas e, dessa forma, a quantidade de memória alocada vai variar em acordo com a quantidade de elementos inseridos na lista (CELES, CERQUEIRA, RANGEL, 2017).

A ideia que permite a construção desse tipo de lista consiste em fazer com que cada nó de uma lista encadeada possua ao menos dois atributos: o valor do nó e uma referência para o próximo nó da lista.

Veja a representação de um nó em uma lista encadeada:



Uma lista simplesmente encadeada com três nós contendo os números inteiros 3, 1 e 2 ficaria assim:

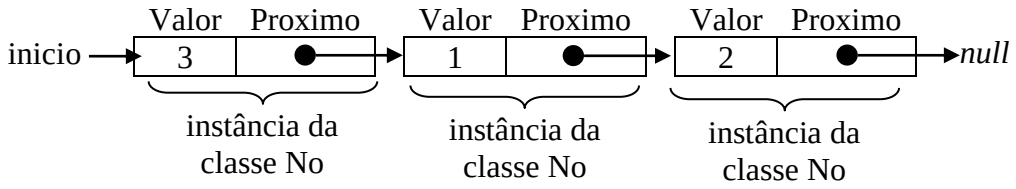


Note que o início da lista aponta para o primeiro nó, sendo que este nó possui o valor 3. O primeiro nó aponta para o segundo nó, que possui o valor 1. Já o segundo nó aponta para o terceiro nó que possui o valor 2. O terceiro nó não aponta para um outro nó, o que indica que não existem elementos após o terceiro elemento da lista.

Em C#, utilizaremos uma propriedade para representar o próximo nó da lista. Essa propriedade terá o tipo da própria classe nó. Se essa propriedade não referenciar um nó, a mesma estará nula, logo o nó cujo próximo seja igual a *null* será o último nó da nossa lista. Veja abaixo o código da classe nó cuja propriedade Valor é do tipo *int*:

```
public class No
{
    public int Valor { get; set; }

    public No Proximo { get => proximo; set => proximo = value; }

    private No proximo = null;
}
```

Com esse código, podemos alterar a representação da nossa lista simplesmente encadeada:

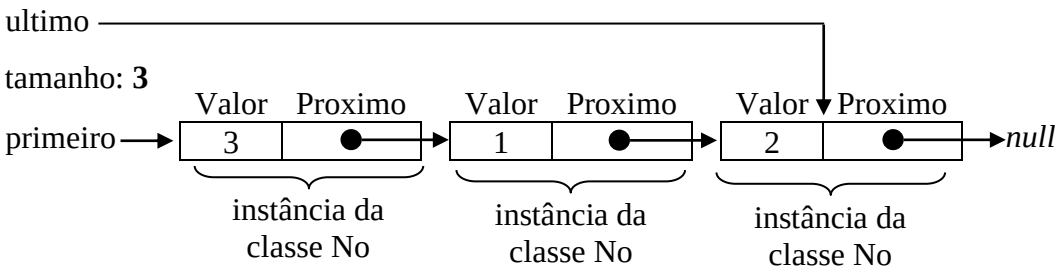


Em acordo com essa representação, segue abaixo o código da classe ListaSimplesmenteEncadeada.

```
public class ListaSimplesmenteEncadeada
{
    public No Primeiro { get => primeiro; set => primeiro = value; }

    public void AdicionarNoInicio(int valor)
    {
        var novoNo = new No { Valor = valor };

        if (primeiro == null)
            primeiro = novoNo;
        else
        {
            novoNo.Proximo = primeiro;
            primeiro = novoNo;
        }
    }

    public void AdicionarNoFinal(int valor)
    {
        var novoNo = new No { Valor = valor };

        if (primeiro == null)
            primeiro = novoNo;
        else
        {
            var atual = primeiro;

            while (atual.Proximo != null)
                atual = atual.Proximo;

            atual.Proximo = novoNo;
        }
    }

    public void RemoverNoInicio()
    {
        if (primeiro != null) primeiro = primeiro.Proximo;
    }

    public void RemoverNoFinal()
    {
        if (primeiro != null)
        {
            if (primeiro.Proximo == null)
                primeiro = null;
            else
            {
                var atual = primeiro;

                while (atual.Proximo.Proximo != null)
                    atual = atual.Proximo;

                atual.Proximo = null;
            }
        }
    }
```

```
        public No Localizar(int valor)
        {
            for (var no = primeiro; no != null; no = no.Proximo)
                if (no.Valor == valor)
                    return no;

            return null;
        }

        private No primeiro = null;
    }
```

Algumas vezes é importante dispor de outras informações sobre a lista. Um exemplo de uma informação importante sobre a lista é o seu tamanho e outro exemplo é a indicação de qual o seu último nó. O conjunto de dados gerais sobre a lista é conhecido por descritor da lista. A representação da lista encadeada apresentada anteriormente, com a adição de descritores, segue abaixo:



Segue o código fonte da nossa lista encadeada com descritores:

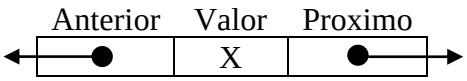
```
    public class ListaEncadeada
    {
        #region Descritores
        public No Primeiro { get => primeiro; }
        public No Ultimo { get => ultimo; }
        public int Tamanho { get => tamanho; }
        #endregion

        public void AdicionarNoInicio(int valor)
        {
            var novoNo = new No { Valor = valor };

            if (primeiro == null)
                primeiro = ultimo = novoNo;
            else
            {
                novoNo.Proximo = primeiro;
                primeiro = novoNo;
            }

            tamanho++;
        }
```

```
public void AdicionarNoFinal(int valor)
{
    var novoNo = new No { Valor = valor };

    if (primeiro == null)
        primeiro = ultimo = novoNo;
    else
    {
        ultimo.Proximo = novoNo;
        ultimo = novoNo;
    }
    tamanho++;
}

public void RemoverNoInicio()
{
    if (primeiro != null)
    {
        if (primeiro == ultimo)
            primeiro = ultimo = null;
        else
            primeiro = primeiro.Proximo;

        tamanho--;
    }
}

public void RemoverNoFinal()
{
    if (primeiro != null)
    {
        if (primeiro == ultimo)
            primeiro = ultimo = null;
        else
        {
            var atual = primeiro;

            while (atual.Proximo.Proximo != null)
                atual = atual.Proximo;

            atual.Proximo = null;
            ultimo = atual;
        }
        tamanho--;
    }
}

public No Localizar(int valor)
{
    for (var no = primeiro; no != null; no = no.Proximo)
        if (no.Valor == valor)
            return no;

    return null;
}
```

```
        private No primeiro = null;
        private No ultimo = null;
        private int tamanho = 0;
    }
```

TÓPICO III – LISTA DUPLAMENTE ENCADEADA

Uma lista duplamente encadeada é uma lista em que cada nó possui duas referências, sendo uma para o nó anterior e outro para o nó seguinte (CELES, CERQUEIRA, RANGEL, 2017). Veja abaixo a representação de um nó em uma lista duplamente encadeada:

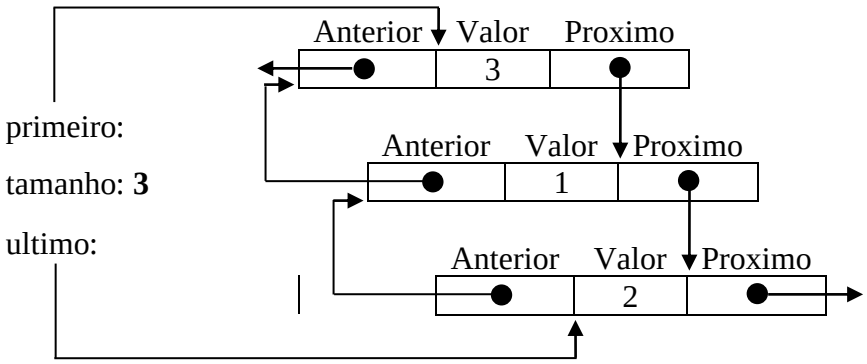


A classe nó para uma lista encadeada de inteiros pode ser codificada da seguinte forma:

```
public class No
{
    public int Valor { get; set; }

    public No Proximo { get; set; }

    public No Anterior { get; set; }
}
```

Em uma lista duplamente encadeada, que geralmente possui descritor, é possível percorrer todos os seus nós não somente do primeiro para o último nó, mas também do último para o primeiro.

Veja a representação de uma lista duplamente encadeada:



Uma das vantagens da lista acima para a lista simplesmente encadeada com descritor é a facilidade de remoção do último nó, pois nessa lista não mais é necessário percorrer todos os nós até o penúltimo de forma que o último possa ser removido. Se a lista possuir mais de um nó, o penúltimo nó pode ser obtido apenas checando o nó anterior ao último (**`ultimo.Anterior`**);

Uma classe que implementa uma lista duplamente encadeada segue abaixo.

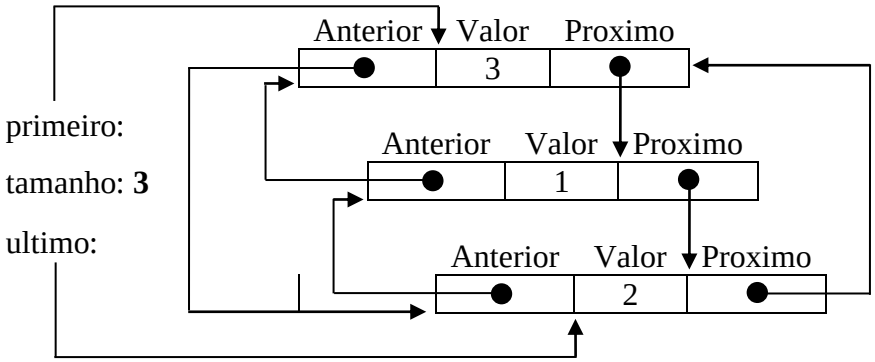
```
public class ListaDuplamenteEncadeada
{
    #region Descritores
    public No Primeiro { get => primeiro; }
    public No Ultimo { get => ultimo; }
    public int Tamanho { get => tamanho; }
    #endregion

    public void AdicionarNoInicio(int valor)
    {
        var novoNo = new No { Valor = valor };

        if (primeiro == null)
            primeiro = ultimo = novoNo;
        else
        {
            primeiro.Anterior = novoNo;
            novoNo.Proximo = primeiro;
            primeiro = novoNo;
        }

        tamanho++;
    }

    public void AdicionarNoFinal(int valor)
    {
        var novoNo = new No { Valor = valor };

        if (primeiro == null)
            primeiro = ultimo = novoNo;
        else
        {
            novoNo.Anterior = ultimo;
            ultimo.Proximo = novoNo;
            ultimo = novoNo;
        }

        tamanho++;
    }
```

```
        public void RemoverNoInicio()
        {
            if (primeiro != null)
            {
                if (primeiro == ultimo)
                {
                    primeiro = ultimo = null;
                }
                else
                {
                    primeiro = primeiro.Proximo;
                    primeiro.Anterior = null;
                }

                tamanho--;
            }
        }

        public void RemoverNoFinal()
        {
            if (primeiro != null)
            {
                if (primeiro == ultimo)
                {
                    primeiro = ultimo = null;
                }
                else
                {
                    ultimo.Anterior.Proximo = null;
                    ultimo = ultimo.Anterior;
                }

                tamanho--;
            }
        }

        public No Localizar(int valor)
        {
            for (var no = primeiro; no != null; no = no.Proximo)
            {
                if (no.Valor == valor)
                {
                    return no;
                }
            }

            return null;
        }

        private No primeiro = null;
        private No ultimo = null;
        private int tamanho = 0;
    }
```

TÓPICO IV – LISTA CIRCULAR

Uma lista circular é uma lista duplamente encadeada em que o último nó está ligado ao primeiro nó (CELES, CERQUEIRA, RANGEL, 2017). Dessa forma, ao navegar no sentido primeiro ao último nó, o próximo nó após o último passa a ser o primeiro nó. Já no sentido último ao primeiro nó, o nó anterior ao primeiro passa a ser o último nó.

Veja abaixo a representação de uma lista circular:



Dependendo de como uma lista duplamente encadeada tenha sido implementada, pode não ser necessária nenhuma alteração na estrutura da lista duplamente encadeada, bastando realizar as duas ligações (próximo e anterior) entre o primeiro e o último nó.

Para a nossa implementação da classe ListaDuplamenteEncadeada, você pode automaticamente atribuir o último elemento da lista como o anterior ao item que estiver sendo inserido e também atribuir o primeiro elemento da lista como sendo o próximo item que estiver sendo inserido. Ao remover itens, você também deverá atualizar essas referências. Veja abaixo o trecho de código que realiza essa ligação entre o primeiro e o último nó:

```
primeiro.Anterior = ultimo;
ultimo.Proximo = primeiro;
```

RESUMO DA UNIDADE III - ATIVIDADES

As listas lineares são estruturas dinâmicas que permitem o armazenamento de dados a serem processados.

Existem diferentes tipos de lista, sendo que neste curso estamos tratando basicamente de dois tipos de listas lineares: listas contíguas e listas encadeadas.

Uma lista contígua aloca espaços sequenciais na memória e, devido a isso, é comum a utilização de vetores para armazenar os elementos de uma lista desse tipo.

Uma lista encadeada não possui a obrigação de alocar espaços de forma sequencial. Para que o armazenamento seja possível, essas listas trabalham como uma estrutura chamada nó. Um nó representa o valor de um elemento da lista assim como também possui uma referência ao próximo nó da lista.

O espaço gasto por uma lista encadeada cresce em acordo com o número de elementos armazenados na mesma.

Algumas informações sobre uma lista encadeada costumam ser controlados nesse tipo de estrutura. Informações como qual o primeiro e o último nó e o tamanho da lista. O conjunto de dados gerais sobre a lista encadeada é conhecido como descritor da lista.

Em uma lista duplamente encadeada, cada nó possui não somente uma referência do próximo nó, mas também uma referência ao nó anterior.

Uma lista circular é uma lista duplamente encadeada onde o próximo nó após o último passa ser uma referência ao primeiro nó e o nó anterior ao primeiro passa a ser uma referência ao último nó.

Exercícios

1) Observe o trecho de código fonte abaixo e selecione a alternativa correta.

```
public void Xxxxx(int valor)
{
    var novoNo = new No { Valor = valor };

    if (primeiro == null)
    {
        primeiro = novoNo;
        ultimo = novoNo;
    }
    else
    {
        novoNo.Proximo = primeiro;
        primeiro = novoNo;
    }
}

public void Yyyyy(int valor)
{
    var novoNo = new No { Valor = valor };

    if (primeiro == null)
    {
        primeiro = novoNo;
        ultimo = novoNo;
    }
    else
    {
        ultimo.Proximo = novoNo;
        ultimo = novoNo;
    }
}
```

e) O procedimento Xxxxx acima remove o primeiro elemento de uma lista simplesmente encadeada. Já o procedimento Yyyyy acima remove o último elemento de uma lista simplesmente encadeada.
f) O procedimento Yyyyy acima remove o primeiro elemento de uma lista simplesmente encadeada. Já o procedimento Xxxxx acima remove o último elemento de uma lista simplesmente encadeada.
g) O procedimento Xxxxx adiciona um novo elemento no início de uma lista simplesmente encadeada. Já o procedimento Yyyyy adiciona um novo elemento no final de uma lista simplesmente encadeada.
h) O procedimento Yyyyy adiciona um novo elemento no início de uma lista simplesmente encadeada. Já o procedimento Xxxxx adiciona um novo elemento no final de uma lista simplesmente encadeada.

2) Em acordo com o trecho de código fonte abaixo, marque a alternativa correta.

```
public void Xxxxx()
{
    if (primeiro != null)
        primeiro = primeiro.Proximo;
}

public void Yyyyy()
{
    if (primeiro != null)
    {
        if (primeiro.Proximo == null)
            primeiro = null;
        else
        {
            var atual = primeiro;

            while (atual.Proximo.Proximo != null)
                atual = atual.Proximo;

            atual.Proximo = null;
        }
    }
}
```

a) O procedimento Xxxxx acima remove o primeiro elemento de uma lista simplesmente encadeada. Já o procedimento Yyyyy acima remove o último elemento de uma lista simplesmente encadeada.
b) O procedimento Yyyyy acima remove o primeiro elemento de uma lista simplesmente encadeada. Já o procedimento Xxxxx acima remove o último elemento de uma lista simplesmente encadeada.
c) O procedimento Xxxxx adiciona um novo elemento no início de uma lista simplesmente encadeada. Já o procedimento Yyyyy adiciona um novo elemento no final de uma lista simplesmente encadeada.
d) O procedimento Yyyyy adiciona um novo elemento no início de uma lista simplesmente encadeada. Já o procedimento Xxxxx adiciona um novo elemento no final de uma lista simplesmente encadeada.

3) Considere uma lista em que cada nó está ligado a um nó anterior e a outro posterior. Nessa mesma lista, se o ultimo nó referenciar o primeiro como o seu próximo nó e o primeiro nó referenciar o último como o seu nó anterior, teremos:

a) Uma lista simplesmente encadeada.
b) Uma lista circular.
c) Uma lista duplamente encadeada que não é circular.
d) Uma lista contígua

4) Veja o trecho de código abaixo e selecione a alternativa que identifica de qual tipo de lista esse trecho foi retirado.

```
public void AdicionarNoFinal(double valor)
{
    valores[qtdeElementos++] = valor;
}

public void RemoverNoFinal(double valor)
{
    qtdeElementos--;
}

private int qtdeElementos = 0;
private const int MAX = 200;
private double[] valores = new double[MAX];
```

a) Lista simplesmente encadeada.
b) Lista duplamente encadeada.
c) Lista circular.
d) Lista contígua.

5) Veja o trecho de código abaixo e selecione a alternativa que identifica de qual o tipo de lista esse trecho foi retirado.

```
public void AdicionarNoInicio(double valor)
{
    valores[--inicio] = valor;
}

public void RemoverNoInicio(double valor)
{
    inicio++;
}

public void AdicionarNoFinal(double valor)
{
    valores[++final] = valor;
}

public void RemoverNoFinal(double valor)
{
    final--;
}

private int inicio = 100;
private int final = 99;
private const int MAX = 200;
private double[] valores = new double[MAX];
```

e) Lista simplesmente encadeada.
f) Lista duplamente encadeada.
g) Lista circular.
h) Lista contígua.

6) O trecho de código abaixo é parte de uma lista circular. Implemente os métodos que adicionam e removem elementos no início e no final da lista.

```
public class ListaCircular
{
    #region Descritores
    public No Primeiro { get => primeiro; }
    public No Ultimo { get => ultimo; }
    public int Tamanho { get => tamanho; }
    #endregion

    private No primeiro = null;
    private No ultimo = null;
    private int tamanho = 0;
}
```

7) Considere a classe ListaCircular da questão 6 e faça um procedimento no método Main da classe Program que receba por parâmetro um objeto da classe ListaCircular e imprima os elementos do primeiro ao último.

Proposta de Gabarito

1) C
2) A
3) B
4) D
5) D
6)

```
public void AdicionarNoInicio(int valor)
{
    var novoNo = new No { Valor = valor };

    if (primeiro == null)
        primeiro = ultimo = novoNo;
    else
    {
        primeiro.Anterior = novoNo;
        novoNo.Proximo = primeiro;
        primeiro = novoNo;
    }

    primeiro.Anterior = ultimo;
    ultimo.Proximo = primeiro;

    tamanho++;
}

public void AdicionarNoFinal(int valor)
{
    var novoNo = new No { Valor = valor };

    if (primeiro == null)
        primeiro = ultimo = novoNo;
    else
    {
        novoNo.Anterior = ultimo;
        ultimo.Proximo = novoNo;
        ultimo = novoNo;
    }

    primeiro.Anterior = ultimo;
    ultimo.Proximo = primeiro;


    tamanho++;
}
```

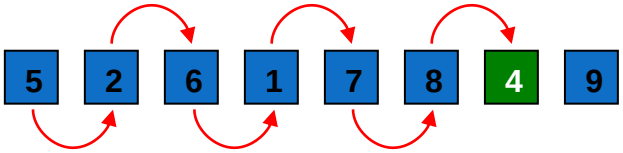
```
        public void RemoverNoInicio()
        {
            if (primeiro != null)
            {
                if (primeiro == ultimo)
                    primeiro = ultimo = null;
                else
                {
                    primeiro = primeiro.Proximo;
                    primeiro.Anterior = null;
                }

                primeiro.Anterior = ultimo;
                ultimo.Proximo = primeiro;

                tamanho--;
            }
        }

        public void RemoverNoFinal()
        {
            if (primeiro != null)
            {
                if (primeiro == ultimo)
                    primeiro = ultimo = null;
                else
                {
                    ultimo.Anterior.Proximo = null;
                    ultimo = ultimo.Anterior;
                }

                primeiro.Anterior = ultimo;
                ultimo.Proximo = primeiro;

                tamanho--;
            }
        }
```

7)

```
    private static void Imprimir(ListaCircular lista)
    {
        var atual = lista.Primeiro;

        Console.Write("Lista: ");

        while (atual != lista.Ultimo)
        {
            Console.Write("{0} ", atual.Valor);

            atual = atual.Proximo;
        }

        Console.WriteLine();
    }
```

UNIDADE II – ÁRVORES

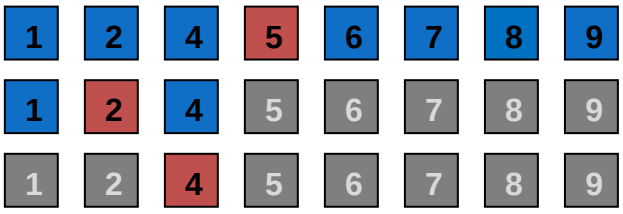
Uma das tarefas comuns em computação consiste na pesquisa de elementos em um determinado conjunto. Uma das estratégias que pode ser utilizada para resolver esse problema é a pesquisa sequencial. Veja abaixo a representação de uma pesquisa sequencial em um vetor.



Note que cada elemento à partir do primeiro é comparado com o valor do elemento que se deseja encontrar. Nesse caso, teremos tempo de busca O(n) e (n + 1) / 2 comparações.

Ainda que se ordene o vetor, o tempo de busca e o número de comparações continuaria o mesmo, pois mesmo que o processamento possa ser concluído assim que o valor seja encontrado ou assim que for encontrado um valor maior que o número a ser pesquisado, ainda assim teríamos uma busca linear e a comparação com a metade dos elementos, quando se considera valores de busca e de elementos aleatórios.

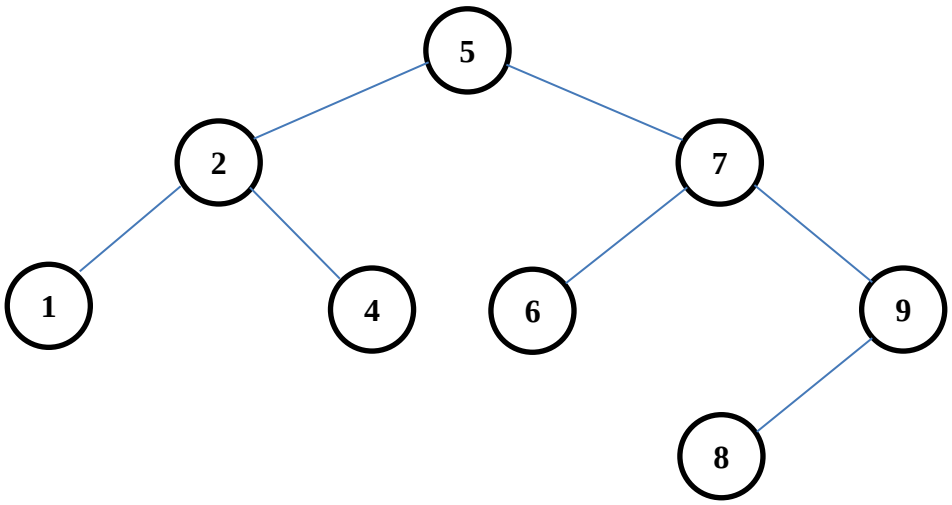
Uma alternativa que apresenta um melhor desempenho é a pesquisa binária. Nessa pesquisa, o vetor precisa estar ordenado e a pesquisa seria iniciada em um elemento que se encontra no meio do vetor. Se o valor procurado for menor que o do elemento que está sendo avaliado, o mesmo processo será feito para os elementos que estão à esquerda do atual. Se o valor for maior, o mesmo processo será feito para os elementos que estão à direita.



O processo se inicia escolhendo o elemento central do vetor, marcado em vermelho. Como o valor desse elemento é 5 e o valor procurado é 4, então a busca deve descartar o 5 e todos os elementos à direita do mesmo para focar a busca nos elementos à direita do 5. Calcula-se o elemento que está no centro do sub-vetor 1, 2, 4 e, como o valor desse elemento é 2, deve-se descartar todos os elementos que estão à esquerda do elemento 2 e o próprio elemento 2. O elemento central do vetor {4} é o próprio 4. Como esse é o valor a ser localizado, o programa pode ser finalizado informando que o elemento foi encontrado.

Note que, na pesquisa binária, a cada comparação feita, o problema diminui pela metade, pois ou os elementos menores ou os elementos maiores que o elemento central do sub-vetor ordenado serão descartados na próxima iteração.

Veja agora essa outra representação do nosso problema de busca.



Nessa representação, vamos chamar de elemento os círculos que possuem os valores. É possível navegar entre um elemento e outro através de suas ligações que chamaremos de arestas.

Com essa representação, podemos realizar a busca navegando entre os nós com a seguinte regra: a busca se inicia de cima para baixo e, sempre que o valor buscado for menor que o valor do nó que está sendo explorado, basta seguir para o próximo nó à esquerda. Se for maior, basta seguir para o próximo nó à direita. Após isso, repete-se o processo até encontrar o elemento ou até acabarem os nós a serem explorados.

A representação que fizemos acima é uma árvore. Seguem alguns problemas que podem utilizar árvores em sua representação: estrutura de dados utilizada pelo Windows Explorer para gerenciar arquivos, representação da hierarquia universitária, representação de expressões aritméticas.

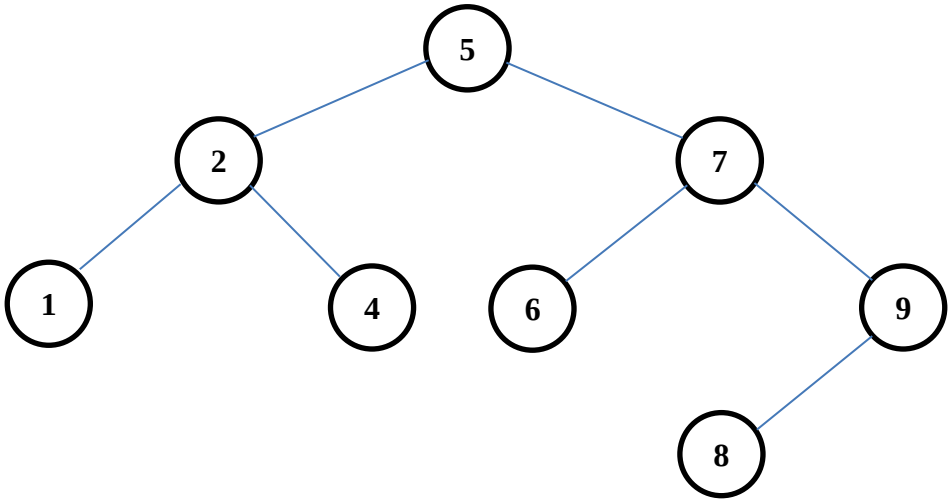
É difícil utilizar listas lineares para representar hierarquias. Já pilhas e filas refletem alguma hierarquia, porém são limitadas a somente uma dimensão. Árvore é uma estrutura criada para superar as limitações de listas lineares, pilhas e filas.

Nesse ponto já podemos iniciar o estudo de árvores. Faremos isso no próximo tópico.

TÓPICO I – REPRESENTAÇÕES

Em computação, uma árvore é formada por nós e arcos, sendo que a raiz é representada no topo e as folhas na base.

A representação abaixo é conhecida por representação hierárquica.



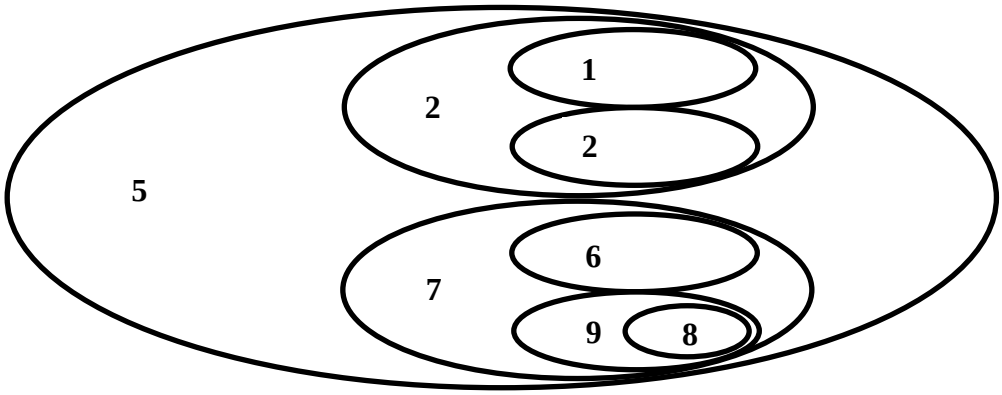
Existe também a representação por alinhamento dos nós. A mesma árvore da figura anterior está representada a seguir. Note que a tabulação identifica a hierarquia dos nós. O número de tabulações de cada nó identifica o nível do nó. A raiz da árvore tem nível zero. Os filhos da raiz tem nível 1, os filhos dos filhos da raiz tem nível 2 e assim sucessivamente.

```
5
        2
                1
                4
        7
                6
                9
                        8
```

Outra representação utilizada é a por parênteses aninhados, conforme abaixo. Note que cada nó é representado dentro de parênteses, sendo que primeiro vem o valor do nó e depois os filhos desse nó.

(5 (2 (1)(4)) (7 (6) (9 (8))))

A última representação que vamos apresentar é a representação por diagramas de inclusão. Veja-a abaixo.



Utilizaremos prioritariamente a representação hierárquica nesse material.

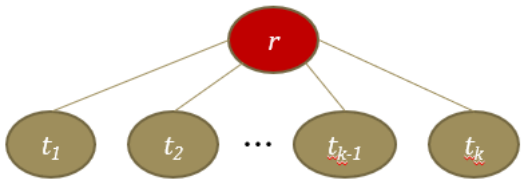
TÓPICO II – DEFINIÇÕES IMPORTANTES

Algumas definições são importantes para a compreensão desse material. Seguem abaixo cada um desses conceitos básicos.

Um nó é um elemento que contém a informação enquanto um arco liga dois nós. Diz-se nó pai o nó superior de um arco e nó filho o nó inferior desse mesmo arco. O nó raiz é aquele que se encontra no topo da árvore, sendo que o nó raiz não possui ancestral (pai), mas pode possuir filhos. Já os nós folhas são nós da extremidades inferiores que não possuem descendentes (filhos), de forma que os seus filhos são uma estrutura vazia.

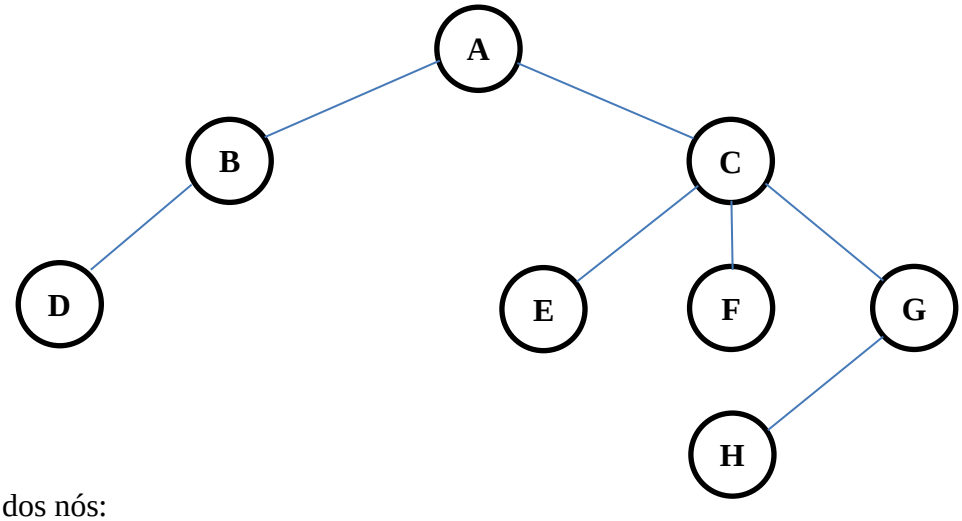
Definição recursiva de árvore (SWARCFITER, MARKENZON, 1994):

1. Uma estrutura vazia é uma árvore vazia.
2. Se t_1, ..., t_k são raízes de árvores disjuntas, então a estrutura cuja raiz r tem como suas filhas as raízes t_1, ..., t_k também é uma árvore.
3. Somente estruturas geradas pelas regras 1 e 2 são árvores.



Grau de um nó é o número de subárvores de um nó. O grau de uma árvore, também conhecido por aridade, corresponde ao grau máximo entre todos os nós de uma árvore (CELES, CERQUEIRA, RANGEL, 2017).

A figura abaixo apresenta a árvore que será utilizada para exemplificar os próximos conceitos apresentados até o fim deste tópico.



Graus dos nós:

G(A): 2
G(B): 1
G(C): 3
G(D): 0
G(E): 0
G(F): 0
G(G): 1
G(H): 0

folhas

Grau da árvore: G(T): 3

Um caminho é uma sequência única de arcos que ligam o nó raiz até outro nó. O comprimento de um caminho corresponde ao número de arcos no caminho.

O nível de um nó é a distância em arcos entre a raiz e o nó em questão. O nível no nó raiz é igual a zero.

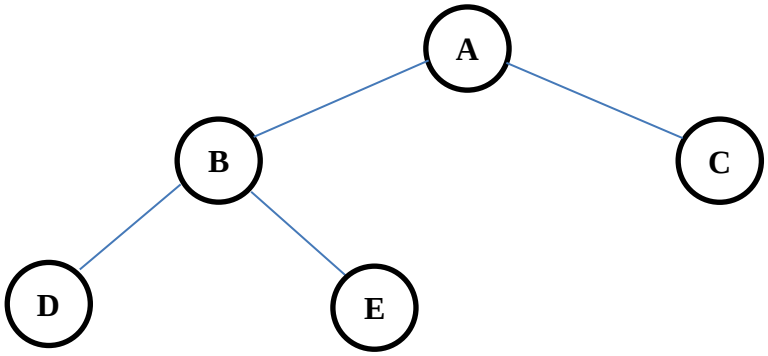
Em nossa árvore de exemplo, possui nível zero o nó A, possuem nível 1 os nós B e C, possuem nível 3 o nó H e todos os outros nós possuem nível 2.

A altura de um nó é a distância entre este e o seu descendente mais afastado, ou seja, o maior caminho entre o nó em questão e um nó folha. A altura da árvore corresponde à altura do nó folha que tem o caminho mais longo até a raiz. Note que uma árvore com um único nó possui altura 0 e uma árvore vazia possui altura -1 por definição.

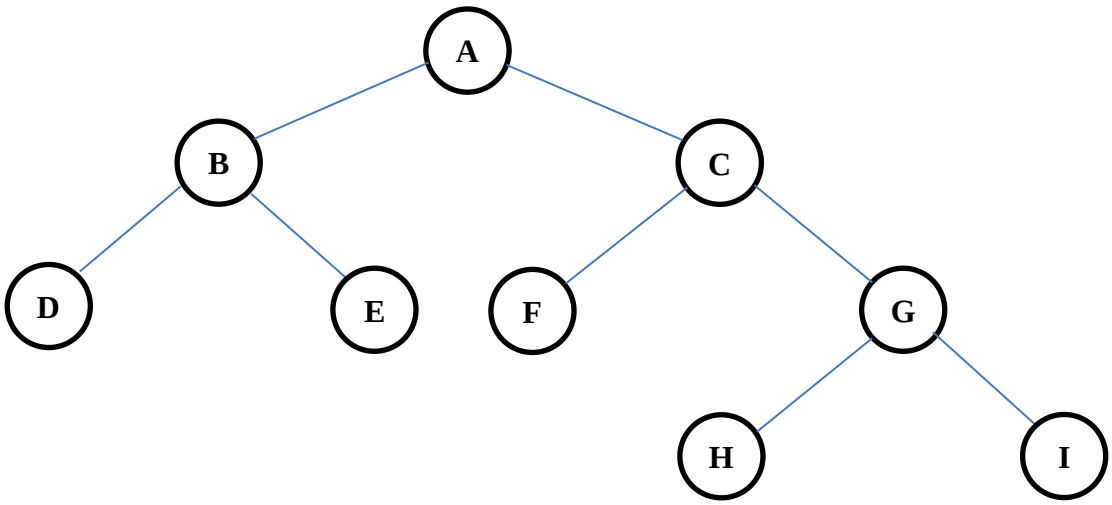
TÓPICO III – ÁRVORE BINÁRIA

Se uma árvore possui grau 2, ou seja, seus nós tem no máximo dois filhos, essa é uma árvore binária. Em uma árvore binária, os nós filhos são chamados de nó filho à esquerda e nó filho à direita (SWARCFITER, MARKENZON, 1994).

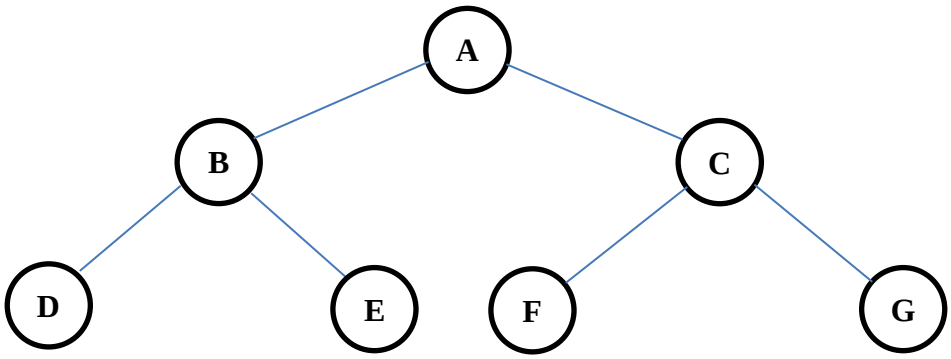
Uma árvore estritamente binária é uma árvore em que cada nó possui nenhum ou dois filhos. Veja o exemplo de uma árvore estritamente binária abaixo.



Uma árvore binária é dita completa quando nós que não possuem grau 2 (2 filhos) estão somente no último ou no penúltimo nível da árvore.



Uma árvore binária é dita cheia quando todos os nós folha se encontram no último nível da árvore. Exemplo de uma árvore binária cheia:



Veja o número de nós para cada nível da árvore acima:

Nível 0: 1
Nível 1: 2
Nível 2: 4

É fácil perceber que uma árvore cheia de nível máximo igual a 3 possui 8 nós no nível 3, pois existem 2 nós filhos para cada um dos 4 nós do nível 2. Logo, o número de nós do nível i de uma árvore binária cheia é igual a 2^i.

A visita sistemática a cada um dos nós de uma árvore, é denominado percurso. Existem diferentes algoritmos para percurso em árvores e é fundamental conhecer o desempenho e a aplicação dos principais algoritmos para percursos em árvores, de forma a possibilitar um melhor desempenho de programas que utilizam tais estruturas.

O que se chama de visitar um nó consiste em operar com o valor do nó. O percurso em árvore consiste em visitar cada nó somente uma vez, ainda que seja permitido passar pelo nó mais de uma vez.

Nesse material, vamos trabalhar com dois tipos de percursos: percurso em profundidade e percurso em extensão.

Para o percurso em profundidade, você deve escolher se visitará primeiramente os filhos à esquerda ou à direita de cada nó. Para exemplificar o percurso em profundidade, vamos assumir que tomaremos primeiramente o percurso à esquerda. Deve-se então iniciar pela raiz e seguir para os filhos à esquerda o quanto for possível. Assim que atingir uma filha, deve-se voltar um nível e seguir para o filho à direita, repetindo o processo de seguir para os filhos à esquerda o quanto for possível.

As três operações que compõem os nossos algoritmos de percurso são: visitar nó (V), percorrer a subárvore da esquerda (L) e percorrer a subárvore da direita (R).

No percurso em profundidade pode ser feito seguindo qualquer uma das seguintes ordens: VLR, LVR, LRV, VRL, RVL, RLV.

Os percursos VLR e VRL são conhecidos como percursos em pré-ordem, também conhecidos por percursos pré-fixados. Já os percursos LVR e RVL são conhecidos por percursos em ordem simétrica ou ordem central ou ainda percursos in-ordem. Os percursos LRV e RLV são percursos em pós-ordem, também conhecidos por percursos pós-fixados.

Segue o passo a passo para o percurso em pré-ordem:

a. Visite a raiz.
b. Percorra a subárvore da esquerda caso VLR ou percorra a subárvore da direita caso (VRL) em pré-ordem.
c. Percorra a subárvore da direita caso VLR ou percorra a subárvore da esquerda caso (VRL) em pré-ordem.

Segue o passo a passo para o percurso em in-ordem:

a. Percorra a subárvore da esquerda caso LVR ou percorra a subárvore da direita caso (RVL) em in-ordem.
b. Visite a raiz.
c. Percorra a subárvore da direita caso LVR ou percorra a subárvore da esquerda caso (RVL) em in-ordem.

Segue o passo a passo para o percurso em pós-ordem:

a. Percorra a subárvore da esquerda caso LRV ou percorra a subárvore da direita caso (RLV) em pré-ordem.
b. Percorra a subárvore da direita caso LRV ou percorra a subárvore da esquerda caso (RLV) em pré-ordem.
c. Visite a raiz.

Outra possibilidade de percurso é conhecido como percurso em extensão ou percurso em largura. Nesse percurso, os nós de um determinado nível são visitados em sequência e, assim que todos os nós daquele nível são visitados, os nós de um próximo nível são visitados. Nesse percurso, os nós são visitados a partir do nível mais baixo até o nível mais alto, ou ainda são visitados do nível mais alto para o nível mais baixo.

Para implementação desse algoritmo, é comum a utilização de uma fila. O passo a passo para um percurso em extensão que percorre a árvore de cima para baixo e da esquerda para a direita será apresentado abaixo.

a. Adicione o nó raiz na fila
b. Visite o nó que se encontra no início da fila.
c. Retire o nó que se encontra no início da fila.
d. Adicione na fila o filho à esquerda do nó retirado no item c (se houver).
e. Adicione na fila o filho à direita do nó retirado no item c (se houver).

TÓPICO IV – ÁRVORES BALANCEADAS

Uma árvore é dita balanceada quando a mesma minimiza o número de comparações no pior caso, considerando que as chaves possuem a mesma probabilidade de ocorrência. Uma árvore balanceada é uma árvore completa, logo, para manter essa característica em aplicações dinâmicas, as operações de inserção e remoção de nós nesse tipo de árvore costumam ser mais complexas. Para ser mais específico, podemos indicar que a complexidade da busca em uma árvore balanceada é O(log n) (SWARCFITER, MARKENZON, 1994).

Uma árvore balanceada bastante conhecida e estudada chama-se AVL. Uma árvore binária é uma árvore AVL quando, para qualquer nó desta árvore, as alturas de suas subárvores fica entre -1 e 1.

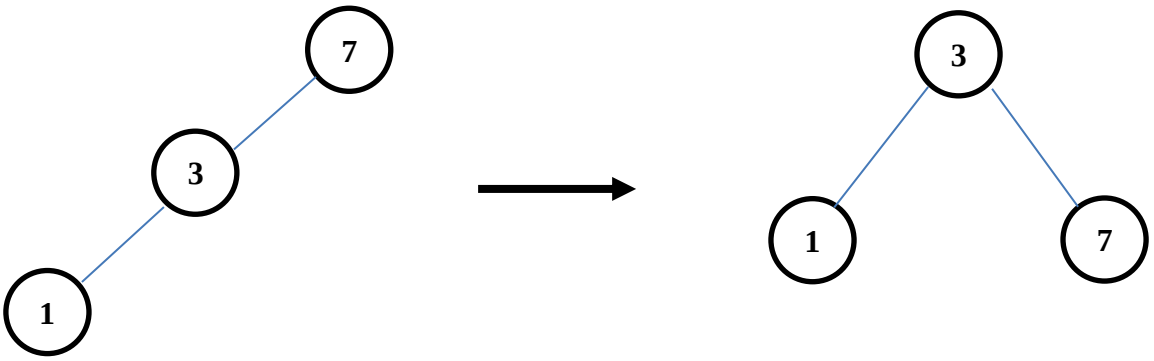
A altura de uma árvore AVL é O(log n), pois é uma árvore balanceada. O custo das operações de inserção e remoção em uma árvore AVL também é O(log n), pois é necessário garantir o balanceamento da árvore após essas operações. O algoritmo de busca em uma árvore AVL é o mesmo do utilizado em uma árvore binária de busca (SWARCFITER, MARKENZON, 1994).

Existem quatro operações de rotação em árvores AVL: rotação simples à esquerda, rotação simples à direita, rotação dupla à esquerda e rotação dupla à direita.

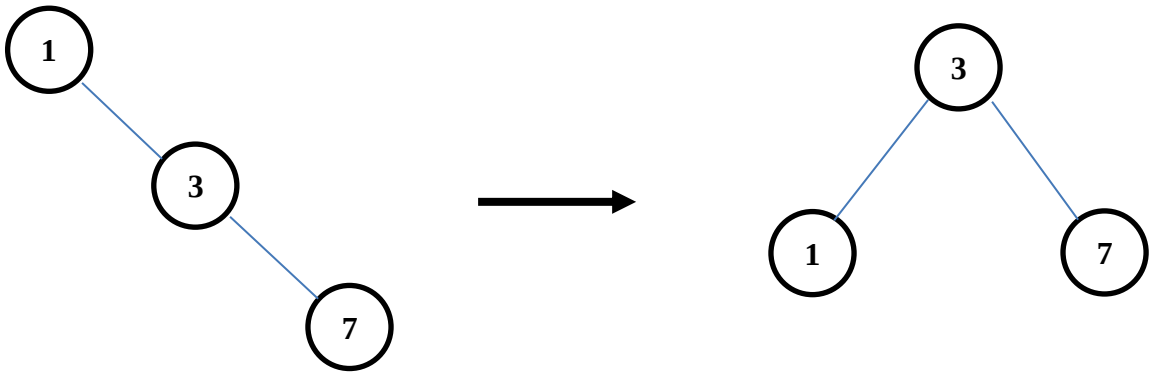
Uma rotação simples deve ser utilizada quando um nó está desbalanceado e o seu filho está no mesmo sentido da inclinação. Já uma rotação dupla deve ser feita quando um nó está desbalanceado e o seu filho está no sentido inverso ao pai, formando um joelho.

Abaixo segue a lista de rotações representadas de forma gráfica.

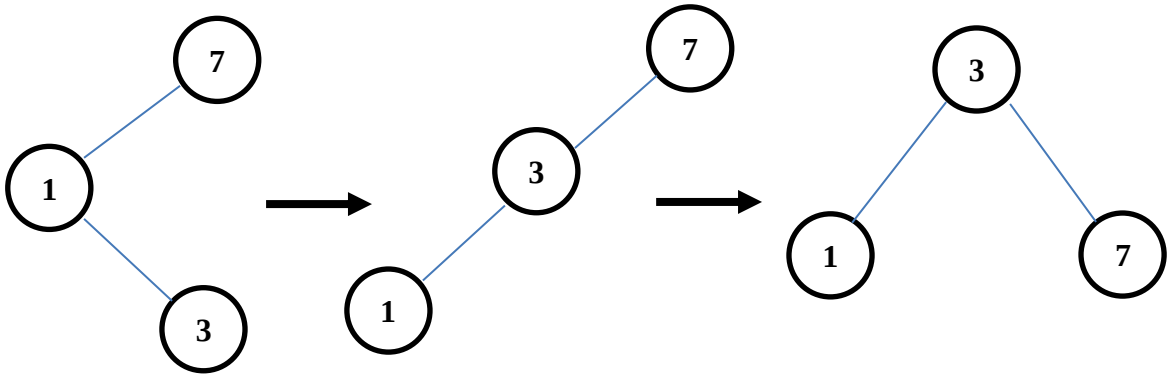
a. Rotação simples à direita:



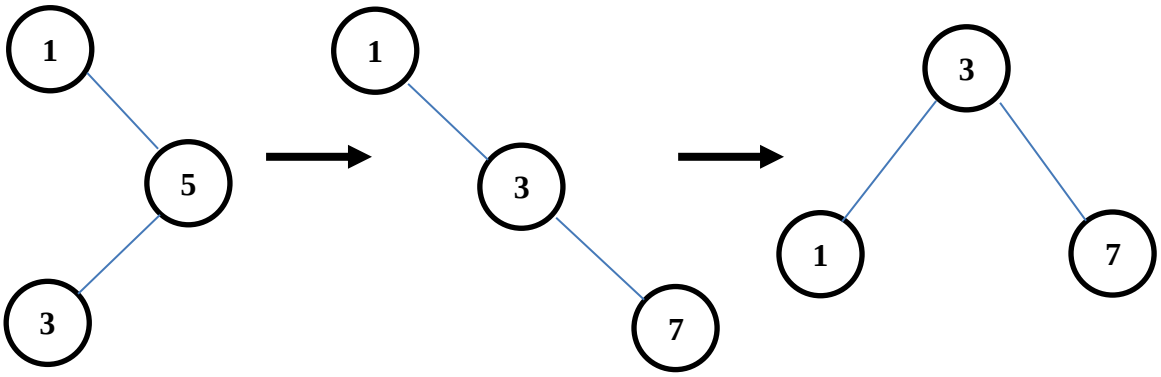
b. Rotação simples à esquerda:



c. Rotação dupla à direita:



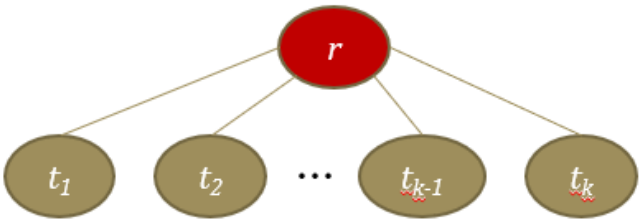
d. Rotação simples à esquerda:



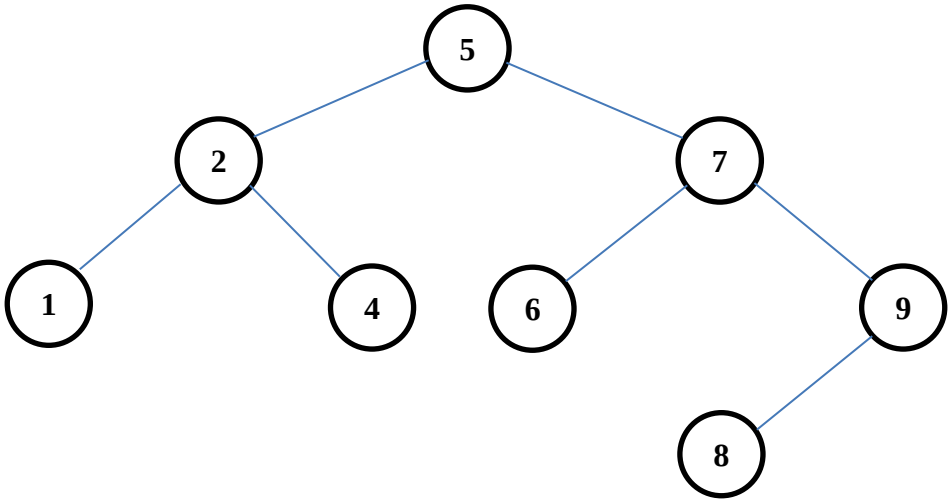
RESUMO DA UNIDADE IV - ATIVIDADES

Definição recursiva de árvore:

1. Uma estrutura vazia é uma árvore vazia.
2. Se t_1, ..., t_k são raízes de árvores disjuntas, então a estrutura cuja raiz r tem como suas filhas as raízes t_1, ..., t_k também é uma árvore.
3. Somente estruturas geradas pelas regras 1 e 2 são árvores.



Existem diferentes representações para árvores, sendo bastante comum a representação hierárquica quando se deseja apresentar graficamente o problema. Segue um exemplo da representação hierárquica:



Outra representação utilizada é a por parênteses aninhados. Note que cada nó é representado dentro de parênteses, sendo que primeiro vem o valor do nó e depois os filhos desse nó. Com essa representação, é fácil criar um programa que leia os dados dessa representação e crie uma árvore em acordo com os mesmos.

(5 (2 (1)(4)) (7 (6) (9 (8))))

Grau de um nó é o número de subárvores de um nó. O grau de uma árvore, também conhecido por aridade, corresponde ao grau máximo entre todos os nós de uma árvore.

Um caminho é uma sequência única de arcos que ligam o nó raiz até outro nó. O comprimento de um caminho corresponde ao número de arcos no caminho.

O nível de um nó é a distância em arcos entre a raiz e o nó em questão. O nível no nó raiz é igual a zero.

A altura de um nó é a distância entre este e o seu descendente mais afastado, ou seja, o maior caminho entre o nó em questão e um nó folha. A altura da árvore corresponde à altura do nó folha que tem o caminho mais longo até a raiz. Note que uma árvore com um único nó possui altura 0 e uma árvore vazia possui altura -1 por definição.

Uma árvore é binária quanto o número máximo de nós filhos de cada um de seus nós é menor ou igual a 2. Logo, um nó de uma árvore binária pode possuir dois filhos: filho à direita e filho à esquerda.

Uma árvore está balanceada quando a mesma é uma árvore completa. As operações de inserção e remoção de nós nesse tipo de árvore costumam ser mais complexas. A complexidade da busca em uma árvore balanceada é O(log n).

A AVL é um tipo de árvore binária balanceada cuja complexidade de inserção e remoção de nós é O(log n).

Exercícios

1) Qual estrutura de dados mais utilizada para permitir o percurso em extensão em uma árvore?

   a) Pilha.

   b) Matriz.

   c) String.

   d) Hash table.

   e) Fila.

2) Qual o número de níveis de uma árvore binária completa com 33 nós?

   a) 3 níveis, variando entre 0 e 2.

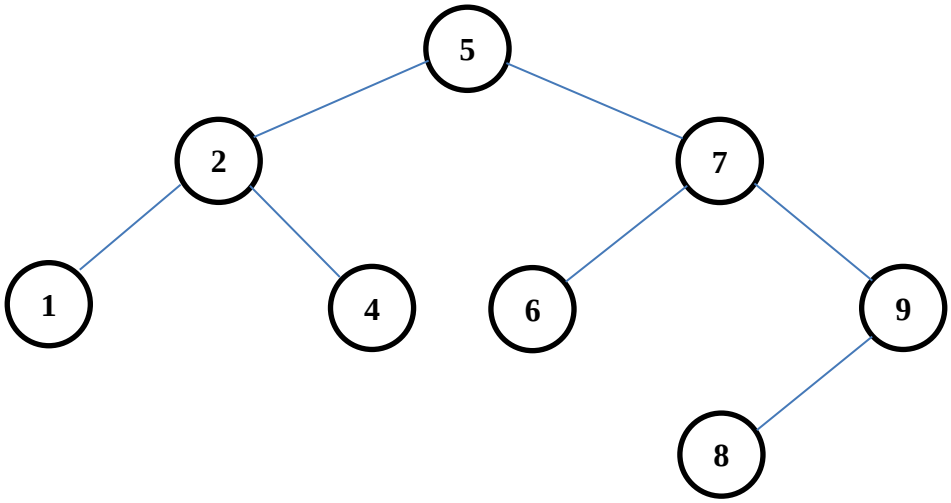
   b) 4 níveis, variando entre 0 e 3.

   c) 5 níveis, variando entre 0 e 4.

   d) 6 níveis, variando entre 0 e 5.

   e) 7 níveis, variando entre 0 e 6.

3) Em acordo com a árvore representada graficamente abaixo, assinale a alternativa correta.



a) Se utilizado o percurso em profundidade VLR, a sequência de visita dos nós seria: 5, 2, 1, 4, 7, 6, 9, 8.

b) Se utilizado o percurso em profundidade LVR, a sequência de visita dos nós seria: 5, 2, 1, 4, 7, 6, 9, 8.

c) Se utilizado o percurso em profundidade LRV, a sequência de visita dos nós seria: 5, 2, 1, 4, 7, 6, 9, 8.

d) Se utilizado o percurso em profundidade VRL, a sequência de visita dos nós seria: 5, 2, 1, 4, 7, 6, 9, 8.

e) Se utilizado o percurso em profundidade RLV, a sequência de visita dos nós seria: 5, 2, 1, 4, 7, 6, 9, 8.

f) Se utilizado o percurso em profundidade RVL, a sequência de visita dos nós seria: 5, 2, 1, 4, 7, 6, 9, 8.

4) Sobre as árvores AVL, assinale a alternativa correta.

   a) Uma árvore de grau 3 pode ser uma AVL.

   b) Cinco é o nível máximo de uma árvore AVL.

   c) A AVL é uma árvore completa.

   d) Para ser AVL, a árvore precisa ser uma árvore cheia.

   e) Uma árvore AVL é projetada de forma a possuir uma complexidade de busca linear.

5) Veja o trecho de código abaixo e responda a alternativa correta.

```
class No
{
   public int Chave { get; set; }
   public No FilhoEsquerda { get; set; }
   public No FilhoDireita { get; set; }
}
```

   a) A classe No apresentada só pode pertencer a uma árvore AVL.

   b) A classe No apresentada pertence a uma árvore de grau 3.

   c) A classe No pode pertencer a uma árvore binária.

   d) Não é possível utilizar a busca em profundidade em uma árvore que utilize a classe No apresentada.

   e) Não é possível utilizar a busca em largura em uma árvore que utilize a classe No apresentada.

6) Represente graficamente a árvore (A (B (D) (E)) (C (F) (G (H)))).

7) Considere a árvore representada graficamente na questão 4 e responda às perguntas abaixo.

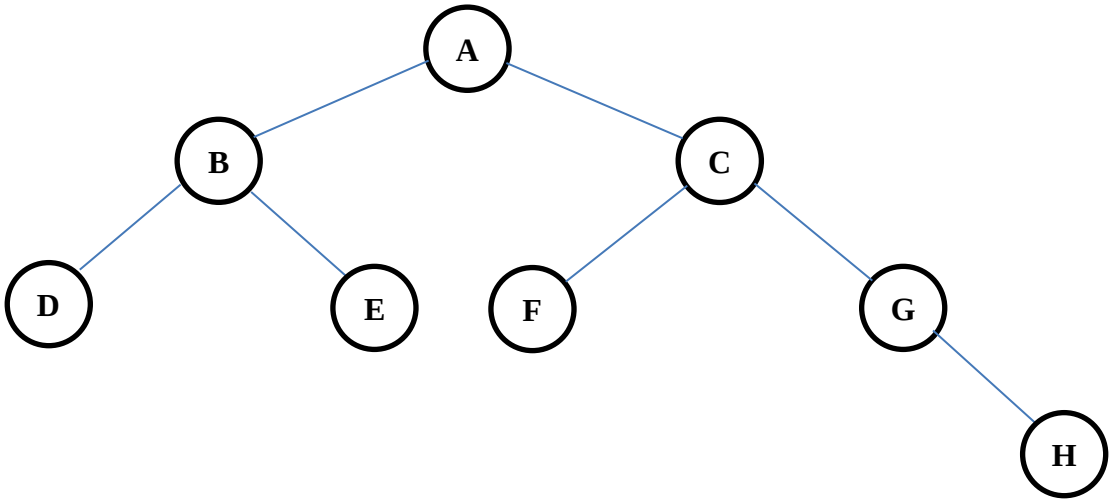
   a) Qual o nível do nó H?

   b) Qual o grau do nó F?

   c) Qual o grau da árvore?

Proposta de Gabarito

1) E
2) C
3) A
4) C
5) C
6)



7)

a) O nível do nó H é 3.
b) O grau do nó F é igual a 0.
c) O grau da árvore é 2.

REFERÊNCIAS

SWARCFITER, J. L., MARKENZON, L. ***Estruturas de Dados e Seus Algoritmos*** *2 ed. Rio de Janeiro: LTC, 1994.*

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José. ***Introdução a Estruturas de Dados: com técnicas de programação em C****. Elsevier Brasil, 2017.*

GOODRICH, Michael T.; TAMASSIA, Roberto. ***Estruturas de Dados e Algoritmos em Java****. 4ª ed. Porto Alegre: Bookman, 2011.*

ASCENCIO, Ana F. G.; CAMPOS, Edilene. ***Fundamentos da Programação de Computadores****. 3ª ed. São Paulo: Pearson Education do Brasil Ltda 2012.*