UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO
PROFESSOR GILVAN JUSTINO
ALGORITMOS E ESTRUTURAS DE DADOS

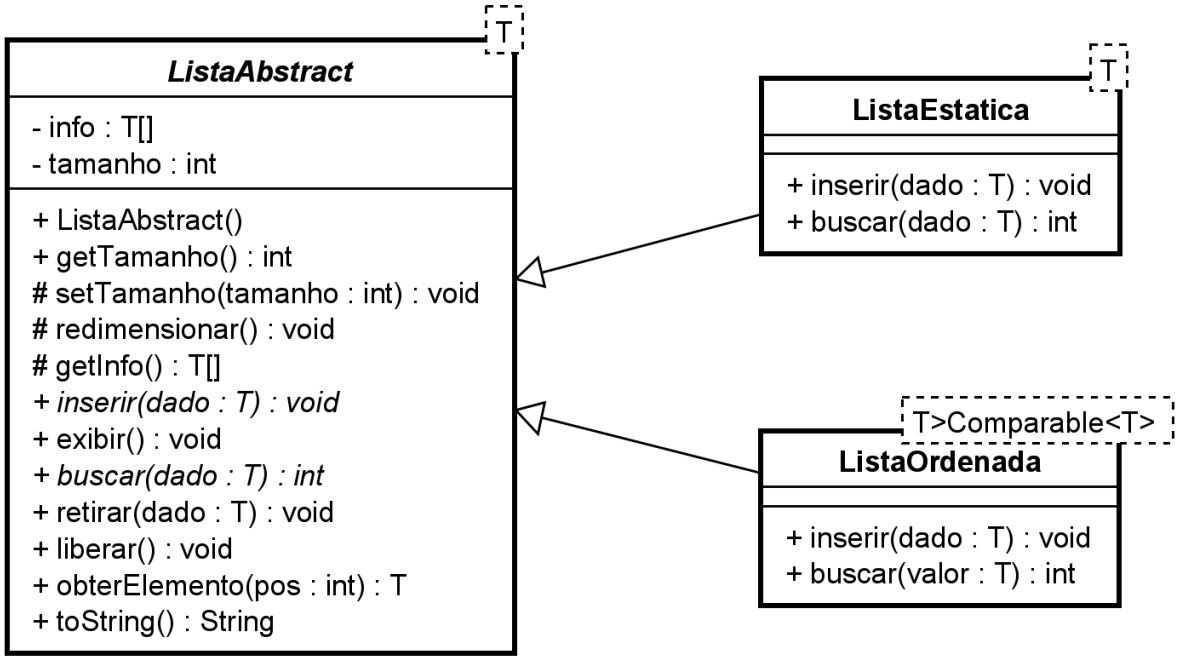
# Lista de Exercício 09

**Questão 1**

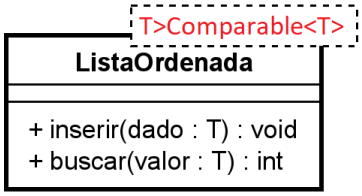
Implementar o algoritmo de busca binária em uma lista com implementação estática, conforme diagrama de classes abaixo.

Para construir a solução, vamos reutilizar a implementação da lista de exercícios 3, como descrito nestas etapas a seguir. Ao final deste exercício, além de manter a classe **`ListaEstatica`**, criaremos uma nova classe, a **`ListaOrdenada`**, que é uma lista estática, porém com a funcionalidade de manter os dados ordenados.

a) Copiar a classe **`ListaEstatica`** do exercício 3 para um novo projeto;
b) Renomear a classe **`ListaEstatica`** para **`ListaAbstract;`**
c) Tornar a classe **`ListaAbstract`** uma classe abstrata;
d) Criar um método *getter* para a variável **`info`**, tornando-o protegido;
e) Criar um método *setter* para a variável **`tamanho`**, tornando-o protegido;
f) Tornar o método **`redimensionar()`** protegido;
g) Criar uma nova classe **`ListaEstatica`**, estendendo-a da classe **`ListaAbstract`**.
h) Tornar os métodos **`inserir()`** e **`buscar()`** da classe **`ListaAbstract`**, métodos abstratos;
i) Mover a implementação do método **`inserir()`** que existia em **`ListaAbstract`** para a nova classe **`ListaEstatica.`** Faça as devidas adaptações na sub-classe;
*Dica:* para acessar ou alterar os dados do vetor **`info`** a partir de subclasses, crie uma variável local para obter a referência da variável. A partir desta referência, acesse o vetor ou altere-o.
j) Mover a implementação do método **`buscar()`** que existe em **`ListaAbstract`** para a nova classe Lista. Faça as devidas adaptações na subclasse.
k) Criar uma classe nova chamada **`ListaOrdenada`**;
l) Implemente o método **`inserir()`** na classe **`ListaOrdenada`** para que adicione um novo dado de forma que a lista permaneça ordenada;
*Dica*: ao criar uma variável na subclasse para armazenar a referência de **`info`** da superclasse, será preciso declarar a variável como um vetor de objetos (ao invés de um vetor de **`T`**)
m) Implementar o método **`buscar()`**na classe **`ListaOrdenada`** utilizando o algoritmo de busca binária.



Observe que a classe **ListaOrdenada** utiliza uma notação diferente da que vimos até agora, para especificar que a classe é parametrizável. Veja destaque a seguir:




A expressão **T>Comparable** tem o seguinte significado: a classe **ListaOrdenada** é uma classe parametrizável, cujo identificador do parâmetro se chama **T**, entretanto, quando o programador declarar uma variável do tipo **ListaOrdenada** deverá submeter como argumento uma classe que implemente a interface **Comparable**. Isto é, quando o programador informar um parâmetro para **ListaOrdenada** não poderá mais informar qualquer classe (como era até então). Com esta declaração, o programador precisará informar uma classe que realize a interface **Comparable**. Como a interface **Comparable** também é parametrizável, vamos fornecer **T** como parâmetro (por isso o **<T>** ao final).

Em Java, a tradução do primeiro compartimento da classe **ListaOrdenada** será assim:

```
public class ListaOrdenada<T extends Comparable<T>> extends ListaAbstract<T>
```

Ou seja, a tradução de **>Comparable<T>** foi introduzir **extends Comparable<T>** imediatamente antes de **>** da declaração do parâmetro de tipo**.** Observação: o uso da palavra reservada **extends**, neste contexto, não representa "herança", mas sim um requisito que pode ser realização ou herança.

Com esta declaração, estamos instruindo o compilador a recusar que o programador submeta qualquer classe como parâmetro para **ListaOrdenada**.

É importante compreender qual a razão de adicionarmos esta restrição. Nos algoritmos de busca, precisamos realizar a comparação entre dois elementos, como ocorre no algoritmo de busca binária, por exemplo:

```
...
se valorBuscar < info[meio] então
    fim ← meio-1; // redefine posição final
...
```

Isto é, temos uma expressão que compara dois itens: **valorBuscar** e **info[meio]**. Considerando que tanto o vetor **info** como a variável **valorBuscar** armazenem referências a objetos, o operador **<** (menor) não é aplicável para compará-los (este operador compara apenas números primitivos). Ora, se os objetos fossem da classe **Aluno** ou da classe **Veiculo**, por exemplo, como poderíamos compará-los, para identificar qual objeto é "menor" que o outro? Aqui é que entra a interface **Comparable**, que possibilita que um objeto seja comparado com outro objeto. Esta interface deve ter sido objeto de estudo na disciplina Programação II. Para quem não lembra, segue referência sobre sua finalidade:

https://docs.oracle.com/javase/9/docs/api/java/lang/Comparable.html

Com isso, a tradução do comando em pseudo-linguagem visto acima, seria em Java desta forma:

```
int comparacao = valor.compareTo((T)info[meio]);
if (comparacao < 0) {
  fim = meio-1;
```

Mais uma observação final: ao usar **getInfo()** para aproveitar o método da superclasse, certifique-se de declarar uma variável de vetor de **Object**, ao invés de T, como em:

```
Object[] info = getInfo();
```

Infelizmente, isso exigirá o uso de *cast* na classe **ListaOrdenada**.

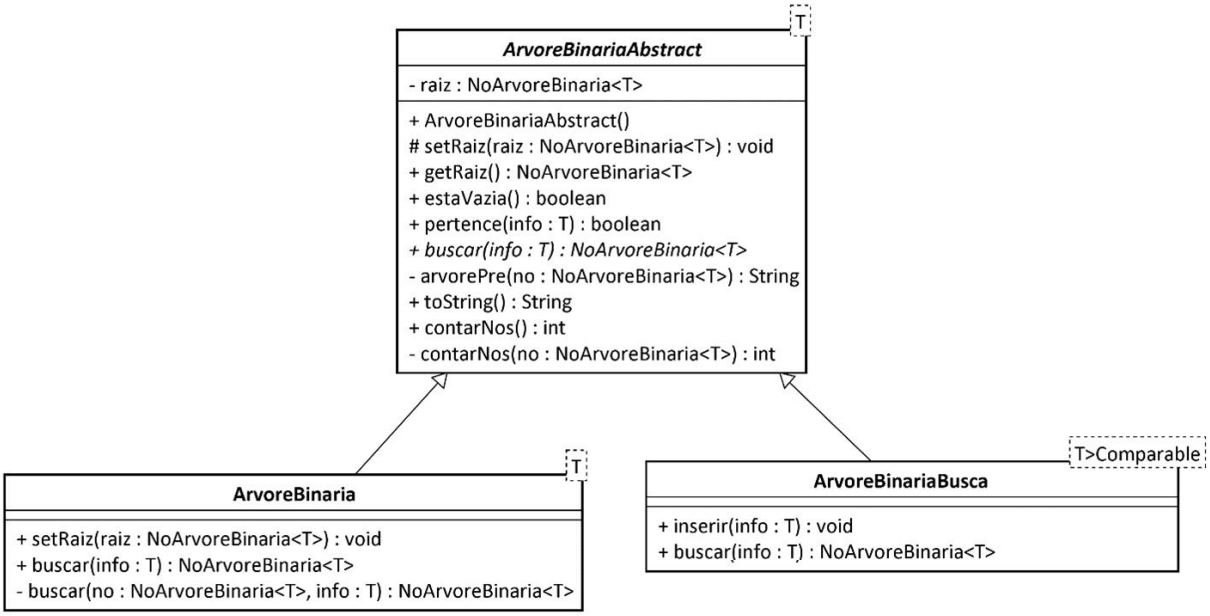
**Questão 2**

Implementar o seguinte plano de testes:

| **Plano de testes PL01 – Validar funcionamento da lista ordenada** | | | |
|---|---|---|---|
| Caso | Descrição | Entrada | Saída esperada |
| 1 | Conferir se o método inserir() mantém os dados ordenados | Criar lista ordenada de números inteiros e adicionar os seguintes dados, nesta ordem: 100, 20, 70, 1 | O método toString() deve retornar 1,20,70,100 |
| 2 | Conferir que o método buscar() localiza um dado armazenado | Criar lista ordenada de números inteiros e adicionar os seguintes dados, nesta ordem: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 | O método buscar(20) deve resultar em 2. |
| 3 | Conferir que o método buscar() localiza um dado armazenado | A entrada é idêntica ao caso 2 | O método buscar(40) deve resultar em 4. |
| 4 | Conferir que o método buscar() localiza um dado armazenado | A entrada é idêntica ao caso 2 | O método buscar(70) deve resultar em 7. |
| 5 | Conferir que o método buscar() localiza um dado armazenado | A entrada é idêntica ao caso 2 | O método buscar(100) deve resultar em 10. |
| 6 | Conferir que o método buscar() localiza um dado armazenado | A entrada é idêntica ao caso 2 | O método buscar(85) deve resultar em -1 |

**Questão 3**

O objetivo desta atividade prática é realizar a implementação de árvores binárias de busca, de acordo com o diagrama de classes da figura abaixo.




Para construir a solução, vamos aproveitar a implementação do exercício 8. Para isso, vamos modificar nossa implementação de árvore binária para que seja extensível. Veja as etapas:

a) Copiar as classes **`ArvoreBinaria`** e **`NoArvoreBinaria`** do exercício 8 para um novo projeto;
b) Renomear a classe **`ArvoreBinaria`** para **`ArvoreBinariaAbstract`**;
c) Tornar a nova classe **`ArvoreBinariaAbstract`** uma classe abstrata;
d) Criar o método abstrato **`buscar()`** na classe **`ArvoreBinariaAbstract`**;
e) Implementar o método **`pertence()`** na classe **`ArvoreBinariaAbstract`** para reusar o método **`buscar()`**;
f) Tornar o método **`setRaiz()`**, da classe **`ArvoreBinariaAbstract`**, protegido;
g) Criar a classe **`ArvoreBinaria`** estendendo-a da classe **`ArvoreBinariaAbstract`**;
h) Inserir o método **`setRaiz()`** na classe **`ArvoreBinaria`** e implementar seu código para reutilizar a implementação do método **`setRaiz()`** da superclasse.
i) Implementar o método **`buscar()`** na classe **`ArvoreBinaria`**, para que localize se há um nó que armazene o dado fornecido como argumento. Em caso positivo, o método **`buscar()`** deve retornar este nó, caso contrário, deverá retornar **`null`**.

Os métodos novos para serem implementados na classe **`ArvoreBinariaBusca`** são:

a) **`inserir()`**: este método deve inserir o dado, fornecido como argumento, na árvore binária de busca. O método deve armazenar o dado num nó de forma que a árvore binária mantenha as características de uma "arvore binária de busca".
b) **`buscar()`**: este método deve buscar o dado fornecido como argumento, na árvore binária, retornando o nó que o armazena. Utilizar o algoritmo de busca binária em árvore.

**Questão 4**

Implemente o seguinte plano de testes.

| **Plano de testes PL01 – Validar funcionamento da implementação de árvore binária de busca** | | | |
|---|---|---|---|
| Caso | Descrição | Entrada | Saída esperada |
| 1 | Conferir se o método inserir() mantém os dados armazenados adequadamente, mantendo a | Criar uma árvore binária de inteiros e adicionar os seguintes dados, nesta ordem:<br>50,30,70,40,25,75,65,35,60 | O método toString() deve retornar:<br><50<30<25<><>><40<35<><>><>><>><70<65<60<><><><><>><75<><>>>> |

| | | | |
|---|---|---|---|
| | árvore com a característica de ser uma árvore binária de busca. | | |
| 2 | Conferir se a árvore consegue remover um nó folha | Criar uma árvore inserindo os dados nesta ordem: 50,30,25,40<br>Remover o nó 40 | O método toString() deve retornar:<br><50<30<25<><><><><> |
| 3 | Conferir se a árvore consegue remover nó com um filho | Criar uma árvore inserindo os dados nesta ordem: 80,52,90,48,71,63,67<br>Remover o nó 71 | O método toString() deve retornar:<br><80<52<48<><><><63<><67<><><><><90<><><> |