---

1. **Conjuntos ordenados e árvores binárias de busca**

   Aprendemos durante o curso que a escolha da estrutura de dados mais apropriada para uma aplicação não é única, e depende de diversos fatores, entre eles:

   - número de elementos armazenados
   - o tipo dos elementos (inteiros, floats, strings, etc.)
   - a escolha das operações de atualização que deseja-se efetuar
   - o tipo de consultas
   - a frequencia das operações

   Nesta prática, nós vamos testar várias soluções para resolver o problema a seguir: dado um conjunto S munido de uma relação de ordem, implemente uma estrutura de dados permitindo preservar S após uma inserção, ou união de S com outro conjunto, por exemplo.

   A seguir encontram-se os arquivos que você precisará nesta prática (contidos no arquivo `pratica7.rar` do SIGAA):

   - `TreeNode.java`: classe representando o nó da árvore binária de busca (a ser completada)
   - `OrderedSet.java`: interface definindo a estrutura de dados abstrata para conjuntos ordenados
   - `BSTSet`: implementação de um conjunto ordenado por árvores binárias de busca
   - `ListNode,OrderedList`: implementação de um conjunto ordenado com listas encadeadas

2. **Conjuntos ordenados genéricos em Java: definição da interface**

   Como mencionado acima, deseja-se comparar várias implementações de um conjunto ordenado; além disso, deseja-se escrever código genérico e portanto facilmente reutilizável permitindo tratar qualquer tipo de dado.

   Para resolver estas duas questões, Java nos fornece duas ferramentas preciosas: as interfaces e as classes genéricas. Na prática, define-se a estrutura de dados abstrata de um conjunto ordenado por meio da interface a seguir, que define as operações de atualização e consulta a serem implementadas.

```
public interface OrderedSet<E extends Comparable<E>> {
  // consultas
  public boolean isEmpty();
  public boolean contains(E e);
```

```
  public E getMin();
  public boolean subset(OrderedSet<E> s);

  // operações de atualização
  public void add(E e);
  public OrderedSet<E> union(OrderedSet<E> s);
}
```

Note que a classe `OrderedSet` é genérica e recebe como parâmetro um tipo `E`, que define os dados que queremos armazenar. Uma vez que trata-se de um conjunto ordenado, é necessário fornecer uma relação de ordem, ou seja, um meio de comparar dois elementos do tipo `E`. Nós vamos exigir que os dados do tipo `E` implementem a interface `Comparable<E>` e disponham de um método `int compareTo(E e)` que efetue a comparação entre o objeto atual (referenciado por `this`) e o objeto `e`. Isto se traduz pelo código Java a seguir:

```
OrderedSet<E extends Comparable<E>>
```

3. **Operações elementares de árvores binárias de busca**

   A fim de representar o conjunto ordenado base de árvores binárias em Java, vamos utilizar a classe `BSTSet`, que encapsula uma referência `root` do tipo `TreeNode` (a raíz da árvore)

```
public class BSTSet<E extends Comparable<E>> implements OrderedSet<E> {

TreeNode<E> root;

  /**
   * Cria uma BST vazia
   */
  public BSTSet() { this.root=null; }

  /**
   * Verifica se um dado elemento já existe
   */
  public boolean contains(E element) {
  return TreeNode.contains(this.root, element);
  }

  /**
   * Retorna o menor elemento do conjunto
   */
  public E getMin() {
```

```
  return TreeNode.getMin(this.root);
  }

...
...
...

}

public class TreeNode<E extends Comparable<E>> {
  final E value;
  final TreeNode<E> left, right;

  public TreeNode(TreeNode<E> left, E value, TreeNode<E> right) {
  //throw new Error("A completar: exercicio 1");
  }

  public TreeNode(E value) {
  //throw new Error("A completar: exercicio 1");
  }

  static<E extends Comparable<E>> boolean contains(TreeNode<E> b, E x) {
  //throw new Error("A completar: exercicio 1");
  }

...
...
...
}
```

O seu objetivo é completar os métodos da classe `TreeNode` que contem:

- um atributo `final E value` que contem o valor armazenado no nó
- dois atributos `final TreeNode<E> left, right` que contem as referências para os nós raízes das subárvores esquerda e direita, respectivamente.

Observações:

Nós queremos representar as árvore de maneira persistente: os atributos acima são declarados como `final`, e portanto podem ser modificados uma única vez (logo de sua utilização pelos construtores da classe `TreeNode`). Se um nó é uma folha, então os atributos `right` e `left` são `null`.

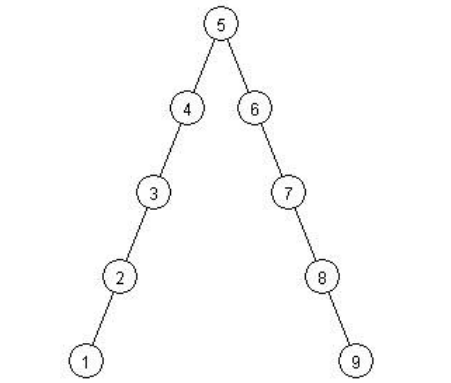
Modifique o arquivo `TreeNode.java` completando-o com os métodos a seguir:

- o construtor `TreeNode(E value)` que constrói uma folha contendo o valor `value`;

- o construtor `TreeNode(TreeNode<E> left, E value, TreeNode<E> right)` que constrói um nó interno, com subárvores esquerda e direita dadas, e contendo o valor `value`;
- o método `contains(TreeNode<E> b, E x)` que retorna `true` se a árvore `b` contem o valor `x`;
- o método `getMin()` que retorna o valor mínimo contido no conjunto. O resultado será `null` se a árvore for vazia;
- o método `add (E e)` que adiciona um novo elemento à árvore. Observação: evitaremos adicionar várias cópias de um valor, logo a árvore não possui valores duplicados.

Para testar o seu código, utilize a função `teste1()` da classe `TestBSTSet`. O resultado a seguir deverá ser obtido:

```
S1=[ 1 2 3 4 5 6 7 8 9]
true
true
true
false
false
min=1
```



4. **Outras operações elementares em árvores binárias de busca**

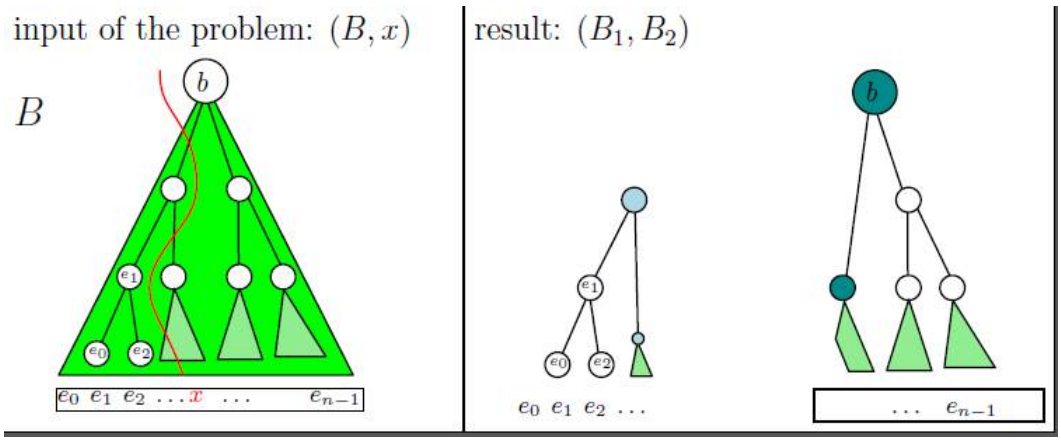
   (a) **União**

   Consideramos agora o problema que consiste em calcular a união de dois conjuntos ordenados (representados sob a forma de árvores binárias). Uma solução simples consiste em transformar as BSTs em duas listas encadeadas através de um percurso em ordem, para em seguida realizar a união de duas listas ordenadas. Esta

solução é simples, tendo como desvantagem o fato de necessitar alocar espaço para a construção das duas listas encadeadas.

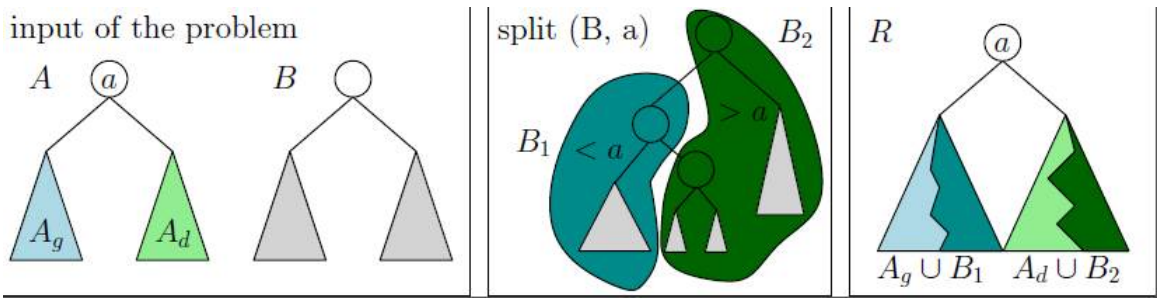
Você irá implementar nesta questão uma solução **mais eficiente** para o problema de união.

O procedimento é recursivo e faz uso de uma função auxiliar (denominada `split`) que decompõe os elementos armazenados de uma árvore em duas partes: a primeira corresponde aos elementos inferiores a um valor x dado, e a segunda parte contem os elementos superiores a este valor.



O procedimento de união (recursivo) é dado abaixo. Denominemos A e B as duas árvores de entrada; seja a o valor da raíz de A, e A_g, A_d suas duas subárvores esquerda e direita:

1. se uma das árvores é vazia, nós retornamos a outra;
2. senão:
   (a) o conjunto B é decomposto ao redor do valor a, obtendo-se duas árvores $B1$ e $B2$
   (b) cria-se recursivamente duas subárvores $E = A_g \cup B1$ e $D = A_d \cup B2$
   (c) retorna-se como resultado a árvore R, tendo a como valor na raíz, e cujas subárvore esquerda e direita são respectivamente E e D

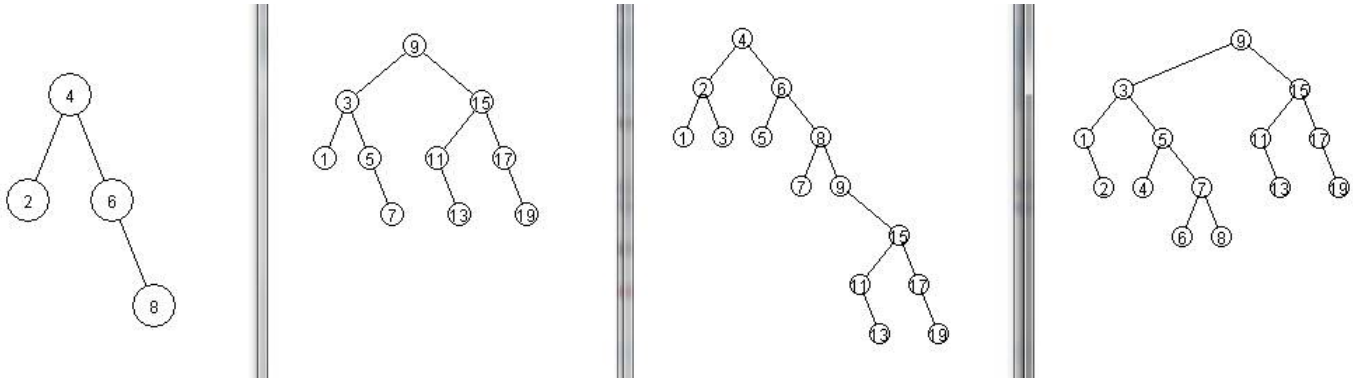


Resta então completar:

- a função `Pair<TreeNode<E>> split(E x, TreeNode<E> s)` que retorna duas novas árvores, contendo respectivamente os elementos que são menores e maiores que o valor x. Observação: você poderá utilizar a classe `Pair<X>` para armazenar (e retornar) o par de árvores binárias calculadas.
- a função `TreeNode<E> union(TreeNode<E> s1, TreeNode<E> s2)` que decompõe a BST através do procedimento descrito acima;

Para testar o seu código, utilize a função `test3()` da classe `TestBSTSet`. Você deverá obter o resultado descrito abaixo. As duas árvores `u1` e `u2` representam o mesmo conjunto obtido pela união das árvores `b1` e `b2`.

```
Testing union
b1: [ 2 4 6 8]
b2: [ 1 3 5 7 9 11 13 15 17 19]
u1:=b1 U b2 = [ 1 2 3 4 5 6 7 8 9 11 13 15 17 19]
u2:=b2 U b1 = [ 1 2 3 4 5 6 7 8 9 11 13 15 17 19]
```



(b) **Testar se um conjunto é subconjunto de outro**

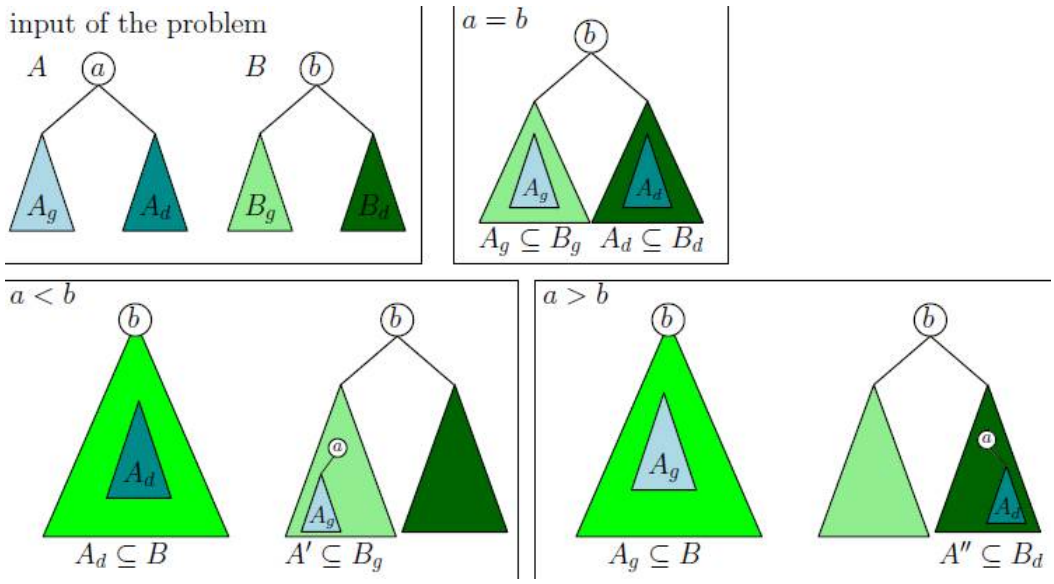
Vamos agora ilustrar um algoritmo recursivo permitindo testar se o conjunto de elementos de uma BST está contido em uma outra BST. Como antes, vamos evitar a solução que transforma BST em listas encadeadas e propor uma solução **mais eficiente**.

Chamemos A e B as duas árvores de entrada; seja a o valor da raíz de A e A_g, A_d suas duas subárvores esquerda e direita (de maneira similar denoratermos b a raíz de B):

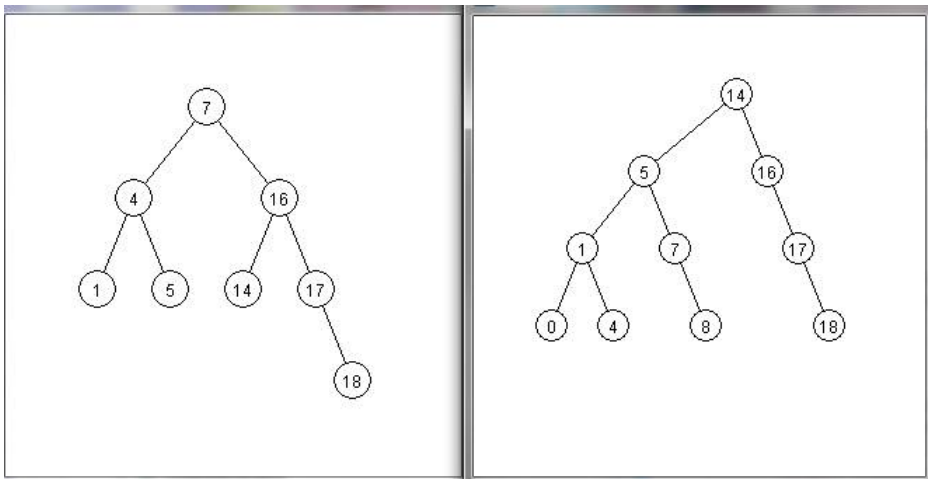
1. se A é vazia, então A é subconjunto de B
2. senão, se B é vazia, a função deve retornar `false`
3. senão, distinguimos 3 casos, de acordo com os valores de a e b - veja na figura a seguir as condições que devem ser verificadas para que A seja subconjunto de B

Você deve completar a função `boolean subset(TreeNode<E> s1, TreeNode<E> s2)` que testa se `s1` é subconjunto de `s2`. Teste o seu código com a função `test4()` da classe `TestBSTSet`.

Você deverá obter o seguinte resultado:




```
Testing subset
b1=[ 1 4 5 7 14 16 17 18]
b2=[ 0 1 4 5 7 8 14 16 17 18]
b1==b2? true
b2==b1? false
```



5. **Uma árvore balanceada a partir de uma lista ordenada**

   Como sabemos, a eficiência dos algoritmos em árvore dependem frequentemente de sua profundidade (igual a distância máxima entre uma folha e a raíz da árvore). Sabemos também que para uma BST sua profundidade depende da ordem de inserção dos elementos: no pior caso, uma ordem pode levar a uma árvore totalmente desequilibrada cuja altura é linear no número de elementos armazenados. Vimos em sala de aula que

existem estruturas de dados que permitem manter árvores bem equilibradas (ex. árvore AVL) a custa de operações de rotação que equilibram a BST dinamicamente.

Se os elementos a serem armazenados puderem ser inseridos na ordem "boa" (por exemplo, se eles já estiverem ordenados em ordem crescente), então é possível construir uma árvore equilibrada com um simples procedimento recursivo. Começamos inicialmente por calcular a profundidade k esperada para a árvore, sendo n o seu número de nós: se ela é equilibrada sua profundidade é dada por $\log_2 n$, o que em Java pode ser calculado por `(int)(Math.log(n) / Math.log(2))`. Em seguida, construimos uma árvore A procedendo da seguinte forma:

1. Se $k = 0$, então a árvore A contem no máximo um nó, e podemos então terminar (perceba que A pode ser também vazia);
2. Senão:
   (a) crie (com uma chamada recursiva), a subárvore G (ela terá tamanho $(n-1)/2$) e profundidade $k-1$)
   (b) crie o nó raíz da árvore A
   (c) crie (com uma chamada recursiva), a subárvore D (de profundidade $k-1$ que deverá conter os elementos restantes)
   (d) retorne a árvore A (tendo G e D como subárvores esquerda e direita, respectivamente

Observação: Este procedimento recursivo efetua a construção da BST de maneira "bottom-up", partindo das folhas.

A entrada do problema (um conjunto ordenado) será representada pela classe genérica `Queue<E>`. Você deve completar a função auxiliar `TreeNode<E> ofList(Queue<E> q, int n, int k)` que constrói uma BST de profundidade k e tamanho n, contendo os elementos de `q`, que será chamado a partir do método `TreeNode<E> ofList(Queue<E> q)` permitindo obter uma árvore equilibrada a partir de uma lista ordenada.

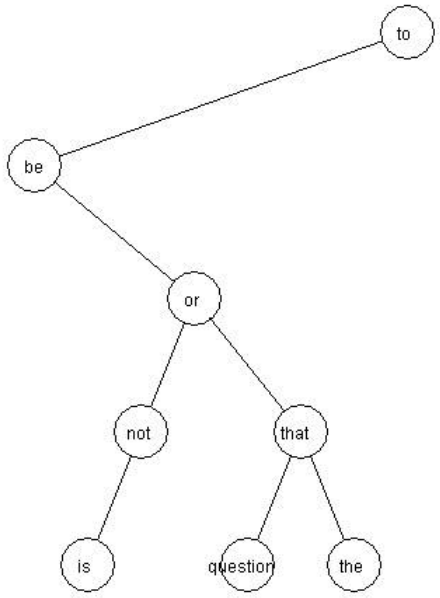
Teste o seu código com a função `test2()` da classe `TestBSTSet`

```
public static void test2() {
  System.out.println("Testing basic operations on binary search trees");
  BSTSet<String> s2=new BSTSet<String>();
  s2.add("to"); s2.add("be"); s2.add("or"); s2.add("not");
  s2.add("to"); s2.add("be"); s2.add("that"); s2.add("is");
  s2.add("the"); s2.add("question");
  System.out.println("Hamlet 3/1: "+s2+"\n");
  new Fenetre(s2.root);

  int n=10;
  Queue<Integer> l = new LinkedList<Integer>();
  for(int i=0;i<n;i++)
```

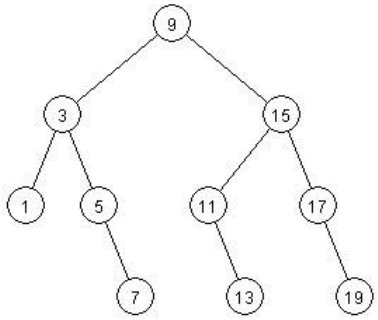
```
      l.add(2*i+1);
    System.out.println(l);
// uma BST a partir de uma lista
    BSTSet<Integer> b2= new BSTSet<Integer>(TreeNode.ofList(l));
    System.out.println(b2);
    new Fenetre(b2.root);
}
```

Você deverá obter o resultado a seguir:

```
Testing basic operations on binary search trees
Hamlet 3/1: [ be is not or question that the to]

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
[ 1 3 5 7 9 11 13 15 17 19]
```



*Este trabalho prático é de autoria de Luca Castelli Aleardi (Poly, France)


9
3
15
1
5
11
17
7
13
19