

BTree / BSTree — spec Nereus + implementación Java

Traducción fiel de la especificación algebraica: misma jerarquía, mismos nombres, misma inmutabilidad observacional, mismas precondiciones, mismos 6 casos del Delete.

- Resumen:** tres archivos Java (BTree.java, BSTree.java, Main.java) más un Makefile. La implementación respeta los mismos cuatro principios de la versión C++:
- **Inmutabilidad observacional** — todos los campos son final; los métodos devuelven nuevas instancias.
 - **Sharing estructural** — los subárboles se guardan por referencia (Node es protected static final con campos final). Cada Insert/Delete crea O(log n) Nodes nuevos y reutiliza el resto.
 - **Encapsulamiento** — el constructor de "ascensión" BSTree(BTree<E>) es protected: refleja la cláusula HIDE BTree_Create. Un cliente externo que llame BTree.BTree_Create(...) recibe un BTree<E> y no puede convertirlo a BSTree<E>.
 - **Precondiciones declaradas** — cada operación parcial chequea su pre: y lanza IllegalStateException con el mensaje exacto del axioma violado.

1. Filosofía de diseño — correspondencia spec ↔ código Java

Construcción Nereus	Traducción Java
CLASS BTree [elem]	public class BTree<E> con generics no-acotados
BASIC CONSTRUCTORS BTree_Empty, BTree_Create	Dos métodos estáticos polimórficos: static <E> BTree<E> BTree_Empty() y static <E> BTree<E> BTree_Create(L, R, v). Representación interna con protected static final class Node<E>; un BTree con node == null es el vacío.
CLASS BSTree [Element:WithOrder]	public class BSTree<E extends Comparable<E>>. WithOrder ≡ Comparable<E>.
IS-SUBTYPE-OF BTree[Element]	extends BTree<E> (herencia pública)
HIDE BTree_Create	El constructor BSTree(BTree<E> t) es protected. Solo el código del mismo paquete (y BSTree mismo) puede invocarlo; un cliente externo no puede ascender un BTree arbitrario a BSTree, garantizando que la invariante BST quede preservada por construcción.
Axiomas ecuacionales	Implementación recursiva con la misma estructura del axioma: if (BTree_IsEmpty()) → caso base; else compara raíz y recurre. Los 6 casos del Delete (A–F) van comentados con su nombre en línea.
Precondiciones pre: ...	throw new IllegalStateException("...precondition violated...")

Inmutabilidad implícita (axiomas son rewriting, no mutación)	Campos <code>final</code> ; valores de retorno son nuevas instancias; Nodes son <code>final</code> dentro de <code>BTree</code>
Sharing estructural natural del término algebraico	Java references — los subárboles inalterados se comparten entre la versión vieja y la nueva (los Nodes son inmutables, así que el sharing es seguro y libre)

2. Mapping operación-por-operación

Nereus	Java	Notas
BTree<E>		
<code>BTree_Empty: -> BTree</code>	<code>static <E> BTree<E> BTree_Empty() o new BTree<>()</code>	Default constructor produce el vacío.
<code>BTree_Create: BTree*BTree*elem -> BTree</code>	<code>static <E> BTree<E> BTree_Create(BTree<E> L, BTree<E> R, E v)</code>	Construye un Node nuevo con sharing.
<code>BTree_IsEmpty</code>	<code>boolean BTree_IsEmpty()</code>	Test <code>node == null</code> .
<code>BTree_Root (parcial)</code>	<code>E BTree_Root()</code>	Lanza <code>IllegalStateException</code> si <code>isEmpty</code> .
<code>BTree_Left / BTree_Right</code>	<code>BTree<E> BTree_Left() / BTree_Right()</code>	Devuelven la referencia del Node (sharing libre).
<code>belongs: elem*BTree -> Boolean</code>	<code>boolean belongs(E e)</code>	Recursivo; usa <code>equals</code> (semántica de igualdad de Java).
BSTree<E extends Comparable<E>> (hereda todo lo de arriba)		
<code>BSTree_Insert pre: not belongs(e, t)</code>	<code>BSTree<E> BSTree_Insert(E e)</code>	Usa <code>e.compareTo(root)</code> para decidir izq/der.
<code>BSTree_Delete pre: belongs(e, t)</code>	<code>BSTree<E> BSTree_Delete(E e)</code>	Implementa los 6 casos A–F con sus comentarios en línea.
<code>BSTree_Min pre: not isEmpty(t)</code>	<code>E BSTree_Min()</code>	Recursión por la izquierda.
Auxiliares Java agregadas (no en spec, para inspección)		
—	<code>int size()</code>	Cantidad de nodos.
—	<code>@Override String toString()</code>	Recorrido in-order como <code>[a, b, c, ...]</code> .
—	<code>String prettyPrint()</code>	Dibujo del árbol con líneas <code> — /</code> <code>└</code> .

3. Implementación Java

java/BTree.java

```
public class BTree<E> {

    class=class="str">"com"> // Nodo interno: representa BTree_Create(left, right, value).
    class=class="str">"com"> // Un BTree con node == null representa BTree_Empty().
    protected static final class Node<E> {
        final BTree<E> left;
        final BTree<E> right;
        final E value;

        Node(BTree<E> left, BTree<E> right, E value) {
            this.left = left;
            this.right = right;
            this.value = value;
        }
    }

    protected final Node<E> node;

    class=class="str">"com"> // — Constructores —————
    public BTree() { this.node = null; }
    protected BTree(Node<E> node) { this.node = node; }
    protected BTree(BTree<E> other) { this.node = other.node; }

    class=class="str">"com"> // — BASIC CONSTRUCTORS de la spec —————
    public static <E> BTree<E> BTree_Empty() {
        return new BTree<>();
    }

    public static <E> BTree<E> BTree_Create(BTree<E> left,
                                           BTree<E> right,
                                           E value) {
        return new BTree<>(new Node<>(left, right, value));
    }

    class=class="str">"com"> // — Observaciones de la spec —————
    public boolean BTree_IsEmpty() { return node == null; }

    public E BTree_Root() {
        if (BTree_IsEmpty())
            throw new IllegalStateException(
                class="str">"BTree_Root: precondition violated (empty tree)");
        return node.value;
    }

    public BTree<E> BTree_Left() {
        if (BTree_IsEmpty())
            throw new IllegalStateException(
                class="str">"BTree_Left: precondition violated (empty tree)");
        return node.left;
    }

    public BTree<E> BTree_Right() {
        if (BTree_IsEmpty())
```

```

        throw new IllegalStateException(
            class="str">"BTree_Right: precondition violated (empty tree)");
        return node.right;
    }

    public boolean belongs(E e) {
        if (BTree_IsEmpty()) return false;
        if (node.value.equals(e)) return true;
        return node.left.belongs(e) || node.right.belongs(e);
    }

    class="class">"str">"com">// — Auxiliares (no en spec, para inspección) —
    public int size() {
        if (BTree_IsEmpty()) return class="num">0;
        return class="num">1 + node.left.size() + node.right.size();
    }

    class="ann">@Override
    public String toString() {
        StringBuilder sb = new StringBuilder(class="str">"[");
        boolean[] first = { true };
        toStringHelper(sb, first);
        sb.append(class="str">"]");
        return sb.toString();
    }

    private void toStringHelper(StringBuilder sb, boolean[] first) {
        if (BTree_IsEmpty()) return;
        node.left.toStringHelper(sb, first);
        if (!first[class="num">0]) sb.append(class="str"> ", ");
        sb.append(node.value);
        first[class="num">0] = false;
        node.right.toStringHelper(sb, first);
    }

    public String prettyPrint() {
        StringBuilder sb = new StringBuilder();
        prettyPrintHelper(sb, class="str"> "", true);
        return sb.toString();
    }

    private void prettyPrintHelper(StringBuilder sb, String prefix, boolean isLast) {
        sb.append(prefix).append(isLast ? class="str">"└─ " : class="str">"├─ ");
        if (BTree_IsEmpty()) { sb.append(class="str"> ".\n"); return; }
        sb.append(node.value).append(class="str"> "\n");
        String newPrefix = prefix + (isLast ? class="str"> "    " : class="str"> "│    ");
        node.left.prettyPrintHelper(sb, newPrefix, false);
        node.right.prettyPrintHelper(sb, newPrefix, true);
    }
}

```

java/BSTree.java

```

public class BSTree<E extends Comparable<E>> extends BTree<E> {

    /** BSTree vacío (BTree_Empty as BSTree). */
    public BSTree() { super(); }
}

```

```

/** Ascensión BTree → BSTree. Protected: refleja HIDE BTree_Create. */
protected BSTree(BTree<E> t) { super(t); }

class=class="str">"com">// BSTree_Insert(t, e) pre: not belongs(e, t)
public BSTree<E> BSTree_Insert(E e) {
    if (belongs(e))
        throw new IllegalStateException(
            class="str">"BSTree_Insert: precondition violated (element already in tree)");
    return insertImpl(e);
}

private BSTree<E> insertImpl(E e) {
    if (BTree_IsEmpty()) {
        return new BSTree<>(BTree_Create(
            new BSTree<E>(), new BSTree<E>(), e));
    }
    E root = BTree_Root();
    BTree<E> L = BTree_Left();
    BTree<E> R = BTree_Right();
    int cmp = e.compareTo(root);
    if (cmp < class="num">0) {
        return new BSTree<>(BTree_Create(
            new BSTree<>(L).insertImpl(e), R, root));
    } else { class="class="str">"com">// cmp > class="num">0 (cmp == class="num">0 excluido por prec
        return new BSTree<>(BTree_Create(
            L, new BSTree<>(R).insertImpl(e), root));
    }
}

class=class="str">"com">// BSTree_Min(t) pre: not BTree_IsEmpty(t)
public E BSTree_Min() {
    if (BTree_IsEmpty())
        throw new IllegalStateException(
            class="str">"BSTree_Min: precondition violated (empty)");
    BTree<E> L = BTree_Left();
    if (L.BTree_IsEmpty()) return BTree_Root();
    return new BSTree<>(L).BSTree_Min();
}

class=class="str">"com">// BSTree_Delete(t, e) pre: belongs(e, t). Casos A-F.
public BSTree<E> BSTree_Delete(E e) {
    if (!belongs(e))
        throw new IllegalStateException(
            class="str">"BSTree_Delete: precondition violated (element not in tree)");
    return deleteImpl(e);
}

private BSTree<E> deleteImpl(E e) {
    E root = BTree_Root();
    BTree<E> L = BTree_Left();
    BTree<E> R = BTree_Right();
    int cmp = e.compareTo(root);

    if (cmp < class="num">0) class="class="str">"com">// Caso A
        return new BSTree<>(BTree_Create(
            new BSTree<>(L).deleteImpl(e), R, root));
    if (cmp > class="num">0) class="class="str">"com">// Caso B
        return new BSTree<>(BTree_Create(
            L, new BSTree<>(R).deleteImpl(e), root));
}

```

```

        class="str">"com">// e == root: casos C, D, E, F
        boolean Lempty = L.BTree_IsEmpty();
        boolean Rempty = R.BTree_IsEmpty();
        if (Lempty && Rempty) return new BTree<>();
        if (Lempty) return new BTree<>(R);
        if (Rempty) return new BTree<>(L);
        class="str">"com">// Caso F.
        E minR = new BTree<>(R).BTree_Min();
        return new BTree<>(BTree_Create(
            L, new BTree<>(R).deleteImpl(minR), minR));
    }
}

```

java/Main.java (demo)

```

public class Main {

    private static <T extends Comparable<T>> void show(String label, BTree<T> t) {
        System.out.println(class="str">"—" + label + class="str">" (size " + t.size() + class="str">")
        System.out.println(class="str">"in-order : " + t.toString());
        System.out.print(t.prettyPrint());
        System.out.println();
    }

    public static void main(String[] args) {
        class="str">"com">// Caso class="num">1 – Delete class="num">10 sobre {class="num">10, class="num">8}
        BTree<Integer> t = new BTree<Integer>()
            .BTree_Insert(class="num">10)
            .BTree_Insert(class="num">8)
            .BTree_Insert(class="num">11);
        show(class="str">"Antes (Insert class="num">10, class="num">8, class="num">11)", t);
        show(class="str">"Delete class="num">10 (Caso F)", t.BTree_Delete(class="num">10));

        class="str">"com">// Caso class="num">2 – Delete class="num">8 sobre {class="num">10, class="num">8}
        show(class="str">"Delete class="num">8 (Caso A→C)", t.BTree_Delete(class="num">8));

        class="str">"com">// Caso class="num">3 – Delete class="num">8 con class="num">7 como hijo
        BTree<Integer> t2 = t.BTree_Insert(class="num">7);
        show(class="str">"Antes (Insert class="num">10, class="num">8, class="num">11, class="num">7)", t);
        show(class="str">"Delete class="num">8 (Caso A→E: class="num">1 hijo)", t2.BTree_Delete(class="num">8));

        class="str">"com">// Inmutabilidad.
        System.out.println(class="str">"t = " + t.toString() + class="str">" (intacto)");

        class="str">"com">// Precondición.
        try { t.BTree_Insert(class="num">5).BTree_Insert(class="num">5); }
        catch (IllegalStateException ex) { System.out.println(ex.getMessage()); }
    }
}

```

4. Compilación y ejecución

```
# Compilar y correr (Java class="num">8+):
cd java
javac -d out BTree.java BSTree.java Main.java
java -cp out Main

# Con el Makefile incluido:
cd java
make run

# Generar un JAR ejecutable (opcional):
cd java/out
jar cfe bstree-demo.jar Main BTree.class BSTree.class BTree\$.class Main.class
java -jar bstree-demo.jar
```

5. Salida esperada del demo

```
— Antes (Insert class="num">10, class="num">8, class="num">11) (size class="num">3) —
in-order : [class="num">8, class="num">10, class="num">11]
└─ class="num">10
   └─ class="num">8
      └─ .
         └─ .
└─ class="num">11
   └─ .
      └─ .

— Delete class="num">10 (Caso F) (size class="num">2) —
in-order : [class="num">8, class="num">11]
└─ class="num">11
   └─ class="num">8
      └─ .
         └─ .
└─ .

— Delete class="num">8 (Caso A→C) (size class="num">2) —
in-order : [class="num">10, class="num">11]
└─ class="num">10
   └─ .
└─ class="num">11
   └─ .
      └─ .

— Antes (Insert class="num">10, class="num">8, class="num">11, class="num">7) (size class="num">4)
in-order : [class="num">7, class="num">8, class="num">10, class="num">11]
└─ class="num">10
   └─ class="num">8
      └─ class="num">7
         └─ .
            └─ .
└─ .
└─ class="num">11
```

```
└ .
└ .
```

— Delete `class="num">8` (Caso A→E: `class="num">1` hijo) (size `class="num">3`) —

in-order : [`class="num">7`, `class="num">10`, `class="num">11`]

```
└ class="num">10
  └ class="num">7
    └ .
      └ .
        └ class="num">11
          └ .
            └ .
```

t = [`class="num">8`, `class="num">10`, `class="num">11`] (intacto)

BSTree_Insert: precondition violated (element already in tree)

6. Diferencias relevantes con la versión C++

Aspecto	C++	Java
Restricción WithOrder	Operadores <, >, == implícitos en el tipo Elem	E extends Comparable<E> explícito; uso de compareTo
Sharing estructural	std::shared_ptr<const Node> con conteo de referencias	Referencias Java + GC + campos final — el sharing es libre y seguro
Encapsulamiento (HIDE)	Constructor private en BSTree; subclase del mismo TU accede	Constructor protected — accesible desde mismo paquete y subclases
Excepción de precondition	std::logic_error con mensaje	IllegalStateException con mensaje idéntico
Comparación de elementos	e == node.value (operador == overloadado)	node.value.equals(e) (semántica explícita de equals)
Templates / generics	Templates: monomorfización en compile-time; tipo concreto en cada instanciación	Generics: type erasure en runtime; un solo BTree compilado
Auxiliares (size, toString, prettyPrint)	Tres métodos en BTree	Idem; toString con @Override para integración natural con println

Sobre HIDE y la visibilidad Java: en Nereus, HIDE BTree_Create dice "el cliente no puede llamar a BTree_Create sobre un BSTree". En Java optamos por *hacer protected el constructor de ascensión* BSTree(BTree<E>) en lugar de cambiar la visibilidad de BTree_Create (que sigue siendo público en BTree — un BTree "puro" lo necesita). Esto es equivalente: cualquier BTree_Create que un cliente arme será de tipo BTree, no BSTree; sin acceso al constructor protegido no puede convertirlo. La invariante BST queda garantizada por construcción.

Sobre el sharing estructural en Java: al hacer `t.BSTree_Insert(8)` sobre un `BSTree` de 100 nodos, solo se crean ~ 7 nuevos `Node` ($\log_2 100$) y un nuevo `BSTree`. El árbol original sigue intacto; los subárboles no tocados se reutilizan a través de las referencias del Java GC. Es la **persistencia barata** típica de estructuras funcionales — $O(\log n)$ tanto en tiempo como en memoria adicional, no $O(n)$.

Sobre `BSTree.equals` y semántica de igualdad de elementos: el `belongs` usa `node.value.equals(e)`, no `==`. Si la spec usa `Integer`, Java autoboxing produce `Integer.equals` que compara por valor; perfecto. Si se usa un tipo custom, hay que asegurar que implemente `equals` y `compareTo` consistentemente (i.e., `compareTo(x) == 0  $\Leftrightarrow$  equals(x)`) para que `belongs` y `insertImpl` sean coherentes.

Paquete: `java/BTree.java`, `java/BSTree.java`, `java/Main.java`, `java/Makefile`. Especificación origen: `BSTree-hide.nereus`. Implementación equivalente: `cpp/` (mismos casos, mismas salidas, mismas trazas).