

BTree / BSTree — spec Nereus + implementación C++

Traducción "fidel" de la especificación algebraica: misma jerarquía (BSTree IS-SUBTYPE-OF BTree), mismos nombres de operaciones, misma inmutabilidad observacional, mismas precondiciones.

- Resumen del paquete:** dos archivos de cabecera C++ (BTree.hpp y BSTree.hpp) más un main.cpp que ejerce las trazas ya documentadas. La implementación respeta cuatro principios traídos del spec algebraico:
- **Inmutabilidad observacional** — todas las operaciones son const y devuelven nuevas instancias.
 - **Sharing estructural** — los subárboles se almacenan vía std::shared_ptr<const Node>: cada Insert/Delete crea O(log n) nodos nuevos y reutiliza el resto.
 - **Encapsulamiento** — la "ascensión" de un BTree a BSTree es un constructor *privado* de BSTree: refleja la cláusula HIDE BTree_Create.
 - **Precondiciones declaradas** — cada operación parcial chequea su pre: y lanza std::logic_error si se viola.

1. Filosofía de diseño — correspondencia spec ↔ código

Construcción Nereus	Traducción C++
CLASS BTree [elem]	template <typename Elem> class BTree
BASIC CONSTRUCTORS BTree_Empty, BTree_Create	Dos métodos estáticos: BTree::BTree_Empty() y BTree::BTree_Create(L, R, v); representación interna con std::shared_ptr<const Node> (nullptr = empty).
CLASS BSTree [Element:WithOrder]	template <typename Element> class BSTree. La restricción WithOrder se asume vía operator<, operator>, operator== en Element.
IS-SUBTYPE-OF BTree[Element]	class BSTree : public BTree<Element> (herencia pública).
HIDE BTree_Create	El constructor BSTree(const BTree<Element>&) está marcado private. Solo los métodos de BSTree pueden "ascender" un BTree a BSTree: el cliente no puede saltarse BSTree_Insert para violar la invariante BST.
Axiomas equacionales (cada uno define un caso del operador)	Implementación recursiva que sigue la estructura del axioma: if (isEmpty) → caso base; else compara raíz y recurre.
Precondiciones pre: ...	throw std::logic_error al violarlas, antes de cualquier modificación.
Inmutabilidad implícita (axiomas dan rewriting, no mutación)	Métodos const; valores de retorno por valor; nodos const dentro de shared_ptr.

2. Mapping operación-por-operación

Nereus	C++	Notas
BTree		
BTree_Empty: -> BTree	static BTree BTree_Empty() o BTree()	Default constructor produce el vacío.
BTree_Create: BTree*BTree*elem -> BTree	static BTree BTree_Create(const BTree& L, const BTree& R, const Elem& v)	Construye un nodo nuevo con sharing.
BTree_IsEmpty: BTree -> Boolean	bool BTree_IsEmpty() const	Test node_ == nullptr.
BTree_Root: BTree(t) -> elem <i>pre: not isEmpty(t)</i>	Elem BTree_Root() const	Lanza si isEmpty.
BTree_Left / BTree_Right	BTree BTree_Left() / BTree_Right() const	Devuelven copias (estructuralmente compartidas).
belongs: elem*BTree -> Boolean	bool belongs(const Elem& e) const	Recursivo: matchea el axioma (e=e1) or belongs(t1) or belongs(t2).
BSTree (hereda todo lo de arriba)		
BSTree_Insert <i>pre: not belongs(e, t)</i>	BSTree BSTree_Insert(const Element& e) const	Implementa el caso base + 2 reglas recursivas.
BSTree_Delete <i>pre: belongs(e, t)</i>	BSTree BSTree_Delete(const Element& e) const	Implementa los 6 casos A–F.
BSTree_Min <i>pre: not isEmpty(t)</i>	Element BSTree_Min() const	Recursión por la izquierda.
Auxiliares C++ agregadas (no en spec, sirven para inspección)		
—	int size() const	Cantidad de nodos. Necesaria para los cout del demo.
—	std::string toString() const	Recorrido in-order como [a, b, c, ...].
—	std::string prettyPrint() const	Dibujo del árbol con líneas $\text{—} / \text{—}$.

3. Especificación Nereus

```
-- =====
-- BSTree – Especificación con constructores básicos
```

```

-- BTree_Empty (heredado) y BSTree_Insert (propio).
--
-- Encapsulamiento: BTree_Create se OCULTA con La cláusula HIDE
-- para que La API pública de BSTree exponga solo Los dos
-- constructores básicos (BTree_Empty y BSTree_Insert) y Las
-- operaciones derivadas.
-- =====

CLASS BTree [elem]
BASIC CONSTRUCTORS BTree_Empty, BTree_Create
EFFECTIVE
TYPE BTree
OPERATIONS
  BTree_Empty: -> BTree;
  BTree_Create: BTree * BTree * elem -> BTree;
  BTree_IsEmpty: BTree -> Boolean;
  BTree_Root: BTree(t) -> elem      pre: not BTree_IsEmpty(t);
  BTree_Left: BTree(t) -> BTree    pre: not BTree_IsEmpty(t);
  BTree_Right: BTree(t) -> BTree    pre: not BTree_IsEmpty(t);
  belongs: elem * BTree -> Boolean;

AXIOMS t1, t2: BTree; e, e1: elem;
  BTree_IsEmpty(BTree_Empty()) = True;
  BTree_IsEmpty(BTree_Create(t1, t2, e)) = False;
  BTree_Root (BTree_Create(t1, t2, e)) = e;
  BTree_Left (BTree_Create(t1, t2, e)) = t1;
  BTree_Right (BTree_Create(t1, t2, e)) = t2;
  belongs(e, BTree_Empty()) = False;
  belongs(e, BTree_Create(t1, t2, e1)) =
    (e = e1) or (belongs(e, t1) or belongs(e, t2));

END-CLASS

CLASS BSTree [Element: WithOrder]
IS-SUBTYPE-OF BTree[Element]
BASIC CONSTRUCTORS BTree_Empty, BSTree_Insert
EFFECTIVE
OPERATIONS
  BSTree_Insert: BSTree(t) * Element(e) -> BSTree
    pre: not belongs(e, t);
  BSTree_Delete: BSTree(t) * Element(e) -> BSTree
    pre: belongs(e, t);
  BSTree_Min: BSTree(t) -> Element
    pre: not BTree_IsEmpty(t);

AXIOMS b, b1, b2: BSTree; i, j: Element;

  BSTree_Insert(BTree_Empty() as BSTree, i) =
    BTree_Create(BTree_Empty() as BSTree, BTree_Empty() as BSTree, i) as BSTree;

  (i < j) implies
  BSTree_Insert(BTree_Create(b1, b2, j) as BSTree, i) =
    BTree_Create(BSTree_Insert(b1 as BSTree, i), b2, j) as BSTree;

  (i > j) implies
  BSTree_Insert(BTree_Create(b1, b2, j) as BSTree, i) =
    BTree_Create(b1, BSTree_Insert(b2 as BSTree, i), j) as BSTree;

  BTree_IsEmpty(b1) implies

```

```

BSTree_Min(BTree_Create(b1, b2, i) as BSTree) = i;

not BTree_IsEmpty(b1) implies
BSTree_Min(BTree_Create(b1, b2, i) as BSTree) = BSTree_Min(b1 as BSTree);

-- Caso A: j < i, recurrir en el subárbol izquierdo.
(j < i) implies
BSTree_Delete(BTree_Create(b1, b2, i) as BSTree, j) =
    BTree_Create(BSTree_Delete(b1 as BSTree, j), b2, i) as BSTree;

-- Caso B: j > i, recurrir en el subárbol derecho.
(j > i) implies
BSTree_Delete(BTree_Create(b1, b2, i) as BSTree, j) =
    BTree_Create(b1, BSTree_Delete(b2 as BSTree, j), i) as BSTree;

-- Caso C: j = i y ambos vacíos -> resultado vacío.
(j = i and BTree_IsEmpty(b1) and BTree_IsEmpty(b2)) implies
BSTree_Delete(BTree_Create(b1, b2, i) as BSTree, j) =
    BTree_Empty() as BSTree;

-- Caso D: j = i, solo derecho no vacío -> derecho.
(j = i and BTree_IsEmpty(b1) and not BTree_IsEmpty(b2)) implies
BSTree_Delete(BTree_Create(b1, b2, i) as BSTree, j) = b2 as BSTree;

-- Caso E: j = i, solo izquierdo no vacío -> izquierdo.
(j = i and not BTree_IsEmpty(b1) and BTree_IsEmpty(b2)) implies
BSTree_Delete(BTree_Create(b1, b2, i) as BSTree, j) = b1 as BSTree;

-- Caso F: j = i, ambos no vacíos -> reemplazar raíz por
-- el mínimo del subárbol derecho, y borrar ese mínimo.
(j = i and not BTree_IsEmpty(b1) and not BTree_IsEmpty(b2)) implies
BSTree_Delete(BTree_Create(b1, b2, i) as BSTree, j) =
    BTree_Create(
        b1,
        BSTree_Delete(b2 as BSTree, BSTree_Min(b2 as BSTree)),
        BSTree_Min(b2 as BSTree)
    ) as BSTree;

HIDE BTree_Create

END-CLASS

```

4. Implementación C++

cpp/BTree.hpp

```

class=class="str">"com">// =====
class=class="str">"com">// BTree.hpp – Implementación C++ del CLASS BTree de Nereus.
class=class="str">"com">// Filosofía: inmutabilidad, sharing estructural, constructores
class=class="str">"com">// como métodos estáticos.
class=class="str">"com">// =====
class=class="str">"pre">#pragma once

class=class="str">"pre">#include <memory>
class=class="str">"pre">#include <stdexcept>

```

```

class=class="str">"pre">#include <sstream>
class=class="str">"pre">#include <string>

template <typename Elem>
class BTree {
protected:
    struct Node {
        std::shared_ptr<const BTree<Elem>> left;
        std::shared_ptr<const BTree<Elem>> right;
        Elem value;

        Node(std::shared_ptr<const BTree<Elem>> l,
            std::shared_ptr<const BTree<Elem>> r,
            Elem v)
            : left(std::move(l)), right(std::move(r)), value(std::move(v)) {}
    };

    std::shared_ptr<const Node> node_;

    explicit BTree(std::shared_ptr<const Node> n) : node_(std::move(n)) {}

public:
    class=class="str">"com">// — Constructores básicos (BASIC CONSTRUCTORS) —————
    BTree() : node_(nullptr) {}
    static BTree BTree_Empty() { return BTree(); }

    static BTree BTree_Create(const BTree& left,
                            const BTree& right,
                            const Elem& value) {
        auto n = std::make_shared<const Node>(
            std::make_shared<const BTree<Elem>>(left),
            std::make_shared<const BTree<Elem>>(right),
            value);
        return BTree(n);
    }

    class=class="str">"com">// — Observaciones —————
    bool BTree_IsEmpty() const { return !node_; }

    Elem BTree_Root() const {
        if (BTree_IsEmpty())
            throw std::logic_error(class="str">"BTree_Root: empty tree");
        return node_->value;
    }
    BTree BTree_Left() const {
        if (BTree_IsEmpty())
            throw std::logic_error(class="str">"BTree_Left: empty tree");
        return *(node_->left);
    }
    BTree BTree_Right() const {
        if (BTree_IsEmpty())
            throw std::logic_error(class="str">"BTree_Right: empty tree");
        return *(node_->right);
    }

    bool belongs(const Elem& e) const {
        if (BTree_IsEmpty()) return false;
        return e == node_->value
            || BTree_Left().belongs(e)
            || BTree_Right().belongs(e);
    }

```

```

}

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

std::string toString() const {
    std::ostringstream oss; oss << class="str">"[";
    bool first = true;
    toStringHelper(oss, first);
    oss << class="str">"]"; return oss.str();
}

std::string prettyPrint() const {
    std::ostringstream oss;
    prettyPrintHelper(oss, class="str">"", true);
    return oss.str();
}

private:
    void toStringHelper(std::ostringstream& oss, bool& first) const {
        if (BTree_IsEmpty()) return;
        BTree_Left().toStringHelper(oss, first);
        if (!first) oss << class="str">", ";
        oss << node_->value;
        first = false;
        BTree_Right().toStringHelper(oss, first);
    }

    void prettyPrintHelper(std::ostringstream& oss,
                          const std::string& prefix,
                          bool isLast) const {
        oss << prefix << (isLast ? class="str">"└─ " : class="str">"└─ ");
        if (BTree_IsEmpty()) { oss << class="str">"·\n"; return; }
        oss << node_->value << class="str">"\n";
        std::string newPrefix = prefix + (isLast ? class="str">"    " : class="str">"└─ ");
        BTree_Left().prettyPrintHelper(oss, newPrefix, false);
        BTree_Right().prettyPrintHelper(oss, newPrefix, true);
    }
};

```

cpp/BSTree.hpp

```

class=class="str">"com">// =====
class=class="str">"com">// BSTree.hpp – Implementación C++ del CLASS BSTree de Nereus.
class=class="str">"com">// IS-SUBTYPE-OF BTree → public inheritance.
class=class="str">"com">// HIDE BTree_Create → ascensión BTree→BSTree es constructor privado.
class=class="str">"com">// =====
class=class="str">"pre">#pragma once

class=class="str">"pre">#include class="str">"BSTree.hpp"
class=class="str">"pre">#include <stdexcept>

template <typename Element>
class BSTree : public BTree<Element> {

```

```

private:
    class=class="str">"com">// Ascensión BTree → BSTree. Privado: refleja HIDE BTree_Create.
    explicit BSTree(const BTree<Element>& t) : BTree<Element>(t) {}

public:
    BSTree() : BTree<Element>() {}

    class=class="str">"com">// BSTree_Insert(t, e) pre: not belongs(e, t)
    BSTree BSTree_Insert(const Element& e) const {
        if (this->belongs(e))
            throw std::logic_error(class="str">"BSTree_Insert: element already in tree");
        return InsertImpl(e);
    }

    class=class="str">"com">// BSTree_Min(t) pre: not BTree_IsEmpty(t)
    Element BSTree_Min() const {
        if (this->BTree_IsEmpty())
            throw std::logic_error(class="str">"BSTree_Min: empty tree");
        BTree<Element> L = this->BTree_Left();
        if (L.BTree_IsEmpty()) return this->BTree_Root();
        return BSTree(L).BSTree_Min();
    }

    class=class="str">"com">// BSTree_Delete(t, e) pre: belongs(e, t). Casos A-F.
    BSTree BSTree_Delete(const Element& e) const {
        if (!this->belongs(e))
            throw std::logic_error(class="str">"BSTree_Delete: element not in tree");
        return DeleteImpl(e);
    }

private:
    BSTree InsertImpl(const Element& e) const {
        if (this->BTree_IsEmpty()) {
            return BSTree(BTree<Element>::BTree_Create(
                BTree(), BTree(), e));
        }
        Element root = this->BTree_Root();
        BTree<Element> L = this->BTree_Left();
        BTree<Element> R = this->BTree_Right();
        if (e < root) {
            return BSTree(BTree<Element>::BTree_Create(
                BTree(L).InsertImpl(e), R, root));
        } else { class=class="str">"com">// e > root (e == root excluido por precondition)
            return BSTree(BTree<Element>::BTree_Create(
                L, BTree(R).InsertImpl(e), root));
        }
    }

    BSTree DeleteImpl(const Element& e) const {
        Element root = this->BTree_Root();
        BTree<Element> L = this->BTree_Left();
        BTree<Element> R = this->BTree_Right();

        if (e < root) class=class="str">"com">// Caso A
            return BSTree(BTree<Element>::BTree_Create(
                BTree(L).DeleteImpl(e), R, root));
        if (e > root) class=class="str">"com">// Caso B
            return BSTree(BTree<Element>::BTree_Create(
                L, BTree(R).DeleteImpl(e), root));
    }

```

```

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

```

cpp/main.cpp

```

class="str">"com">// Demo: reproduce las trazas hechas sobre la spec Nereus.
class="str">"pre">#include class="str">"BTree.hpp"
class="str">"pre">#include class="str">"BSTree.hpp"
class="str">"pre">#include <iostream>

template <typename T>
void show(const std::string& label, const BSTree<T>& t) {
    std::cout << class="str">"—" << label << class="str">" (size " << t.size() << class="str">" ) —\n";
    std::cout << class="str">"in-order : " << t.toString() << class="str">"\n";
    std::cout << t.prettyPrint() << class="str">"\n";
}

int main() {
    class="str">"com">// Caso class="num">1 — Delete class="num">10 sobre {class="num">10, class="num">11}
    BSTree<int> t = BSTree<int>()
        .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">11}
    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 izqu
    BSTree<int> t2 = t.BTree_Insert(class="num">7);
    show(class="str">"Antes (Insert class="num">10, class="num">8, class="num">11, class="num">7)", t2);
    show(class="str">"Delete class="num">8 (Caso A→E: class="num">1 hijo)", t2.BTree_Delete(class="num">8));

    class="str">"com">// Inmutabilidad: t no cambia tras Insert.
    std::cout << class="str">"t = " << t.toString() << class="str">" (intacto)\n";

    class="str">"com">// Precondiciones.
    try { t.BTree_Insert(class="num">5).BTree_Insert(class="num">5); }
    catch (const std::logic_error& ex) { std::cout << ex.what() << class="str">"\n"; }

    return class="num">0;
}

```

5. Compilación y ejecución

```
# Con g++ (Linux/macOS/WSL/MinGW):
cd cpp
g++ -std=c++17 -O2 -Wall main.cpp -o bstree_demo
./bstree_demo

# Con clang++:
clang++ -std=c++17 -O2 -Wall main.cpp -o bstree_demo

# Con MSVC (Visual Studio Developer Command Prompt):
cl /std:c++17 /EHsc /W4 main.cpp /Fe:bstree_demo.exe
bstree_demo.exe

# Con el Makefile incluido:
cd cpp
make run
```

6. Salida esperada del demo

— Antes (Insert 10, 8, 11) (size 3) —
in-order : [8, 10, 11]

```
└─ 10
   └─ 8
      └─ .
         └─ .
            └─ 11
               └─ .
                  └─ .
```

— Delete 10 (Caso F) (size 2) —
in-order : [8, 11]

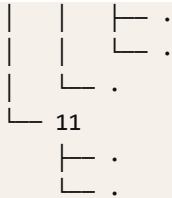
```
└─ 11
   └─ 8
      └─ .
         └─ .
            └─ .
```

— Delete 8 (Caso A→C) (size 2) —
in-order : [10, 11]

```
└─ 10
   └─ .
      └─ 11
         └─ .
            └─ .
```

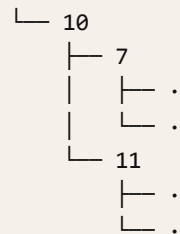
— Antes (Insert 10, 8, 11, 7) (size 4) —
in-order : [7, 8, 10, 11]

```
└─ 10
   └─ 8
      └─ 7
```



— Delete 8 (Caso A→E: 1 hijo) (size 3) —

in-order : [7, 10, 11]



t = [8, 10, 11] (intacto)

BSTree_Insert: element already in tree

Sobre HIDE y la herencia pública: en Nereus, HIDE BTree_Create dice "el cliente no puede llamar a BTree_Create sobre un BSTree". En C++ optamos por *hacer privado el constructor de ascensión* BSTree(const BTree&), en lugar de cambiar la visibilidad de BTree_Create (que sigue siendo público en BTree — un BTree "puro" sí lo necesita). Esto es semánticamente equivalente: cualquier BTree_Create que un cliente arme será de tipo BTree, no BSTree, y no se puede convertir al subtipo. La invariante BST queda garantizada por construcción.

Sobre el sharing estructural: al hacer t.BSTree_Insert(8) sobre un árbol de 100 nodos, solo se crean ~7 nuevos Node ($\log_2 100$). El árbol original sigue intacto y los subárboles no tocados se reutilizan a través de los shared_ptr. Esto es lo que da la persistencia "barata" típica de las estructuras funcionales — pagás $O(\log n)$ por cada Insert/Delete tanto en tiempo como en memoria adicional, no $O(n)$.

Paquete: cpp/BTree.hpp, cpp/BSTree.hpp, cpp/main.cpp, cpp/Makefile. Especificación origen: BSTree-hide.nereus.