

Heap_H — Min-heap como subtipo de BTree (with HIDE)

Constructores básicos públicos: *Heap_Empty* (renamed *BTree_Empty*) y *Heap_Insert*. *BTree_Create* y *Heap_Merge* ocultos.

Idea de diseño: heredando de *BTree* y renombrando *BTree_Empty* como *Heap_Empty*, la única manera que tiene el cliente de construir un *Heap_H* es a través de *Heap_Empty* y *Heap_Insert*. La cláusula *HIDE* *BTree_Create*, *Heap_Merge* oculta el constructor interno y la operación auxiliar de fusión: la propiedad de heap (raíz mínima respecto de los descendientes) queda garantizada por construcción, porque cada axioma de *Heap_Insert* y *Heap_Merge* la preserva.

Implementación interna — skew heap: tanto la inserción como la fusión intercambian los subárboles izquierdo y derecho en cada nivel recursivo. Esto da complejidad amortizada $O(\log n)$ sin necesidad de mantener una métrica de balance explícita (como en heaps leftist). La idea es: cuando se hace recursión sobre un subárbol, ese subárbol "rota" al lado opuesto en el resultado.

1. Especificación

```
-- =====
-- Heap_H — Especificación de min-heap como subtipo de BTree.
--
-- Constructores básicos (Lo que el cliente puede usar):
--   * Heap_Empty   (renombrado de BTree_Empty, heredado de BTree)
--   * Heap_Insert  (propio de Heap_H)
--
-- BTree_Create queda OCULTO mediante HIDE: el cliente solo
-- puede construir heaps componiendo Heap_Empty y Heap_Insert.
-- Esto garantiza por construcción la propiedad de heap (La
-- raíz es siempre menor o igual que todos los descendientes).
--
-- Representación interna: skew heap. Tanto la inserción como
-- la fusión intercambian los subárboles izquierdo y derecho
-- en cada nivel, dando complejidad amortizada  $O(\log n)$  sin
-- necesidad de balanceo explícito.
--
-- Heap_Merge es una operación auxiliar oculta, usada por
-- Heap_DeleteMin para fusionar los dos subárboles tras
-- eliminar la raíz.
-- =====
```

```
CLASS Heap_H [Element : WithOrder]
IS-SUBTYPE-OF BTree[Element]
  rename BTree_Empty as Heap_Empty
BASIC CONSTRUCTORS Heap_Empty, Heap_Insert
EFFECTIVE

OPERATIONS
```

```

-- Renombrado heredado de BTree.
Heap_Empty: -> Heap_H;

-- Constructor propio: inserción que preserva la propiedad de heap.
Heap_Insert: Heap_H(h) * Element(e) -> Heap_H;

-- Observación: el mínimo, que vive siempre en la raíz.
Heap_Min: Heap_H(h) -> Element
    pre: not BTree_IsEmpty(h);

-- Eliminar el mínimo (la raíz).
Heap_DeleteMin: Heap_H(h) -> Heap_H
    pre: not BTree_IsEmpty(h);

-- Operación auxiliar: fusionar dos heaps (skew merge).
-- Se oculta junto con BTree_Create.
Heap_Merge: Heap_H(h1) * Heap_H(h2) -> Heap_H;

AXIOMS h, h1, h2, L, R, L1, L2, R1, R2: Heap_H; x, x1, x2, i: Element;

-- -----
-- Heap_Insert: inserción al estilo skew heap, reescrita a
-- términos con BTree_Create (forma canónica interna).
-- -----

-- Caso base: insertar en heap vacío produce un singleton.
Heap_Insert(Heap_Empty() as Heap_H, i) =
    BTree_Create(Heap_Empty() as Heap_H,
        Heap_Empty() as Heap_H,
        i) as Heap_H;

-- Caso  $i < \text{raíz}$ :  $i$  pasa a ser la nueva raíz; el heap viejo
-- queda íntegro como subárbol izquierdo, el derecho vacío.
-- Se respeta la propiedad de heap:  $i \leq$  todos los descendientes.
( $i < x$ ) implies
Heap_Insert(BTree_Create(L, R, x) as Heap_H, i) =
    BTree_Create(BTree_Create(L, R, x) as Heap_H,
        Heap_Empty() as Heap_H,
        i) as Heap_H;

-- Caso  $i \geq \text{raíz}$ : la raíz  $x$  permanece;  $i$  se inserta
-- recursivamente en el subárbol derecho  $R$ , y luego se
-- intercambian los subárboles (skew) para balance amortizado.
not ( $i < x$ ) implies
Heap_Insert(BTree_Create(L, R, x) as Heap_H, i) =
    BTree_Create(Heap_Insert(R as Heap_H, i), L, x) as Heap_H;

-- -----
-- Heap_Min: el mínimo siempre está en la raíz.
-- -----
Heap_Min(BTree_Create(L, R, x) as Heap_H) = x;

-- -----

```

```

-- Heap_DeleteMin: eliminar la raíz, fusionar L y R.
-- -----
Heap_DeleteMin(BTree_Create(L, R, x) as Heap_H) =
    Heap_Merge(L as Heap_H, R as Heap_H);

-- -----
-- Heap_Merge (skew merge): fusionar dos heaps preservando
-- la propiedad de heap, con intercambio de subárboles en
-- cada nivel recursivo.
-- -----

-- Identidades: fusionar con vacío deja el otro intacto.
Heap_Merge(Heap_Empty() as Heap_H, h) = h;
Heap_Merge(h, Heap_Empty() as Heap_H) = h;

-- Recursivo: el de raíz menor queda arriba; su subárbol
-- derecho se mergea con el otro heap entero; los subárboles
-- izquierdo y derecho se intercambian (skew).

(x1 < x2) implies
Heap_Merge(BTree_Create(L1, R1, x1) as Heap_H,
    BTree_Create(L2, R2, x2) as Heap_H) =
    BTree_Create(
        Heap_Merge(R1 as Heap_H, BTree_Create(L2, R2, x2) as Heap_H),
        L1,
        x1
    ) as Heap_H;

not (x1 < x2) implies
Heap_Merge(BTree_Create(L1, R1, x1) as Heap_H,
    BTree_Create(L2, R2, x2) as Heap_H) =
    BTree_Create(
        Heap_Merge(BTree_Create(L1, R1, x1) as Heap_H, R2 as Heap_H),
        L2,
        x2
    ) as Heap_H;

-- -----
-- Encapsulamiento: ocultar la representación interna.
-- El cliente solo dispone de Heap_Empty y Heap_Insert
-- como constructores; BTree_Create y Heap_Merge son
-- detalles de implementación.
-- -----

HIDE BTree_Create, Heap_Merge

```

END-CLASS

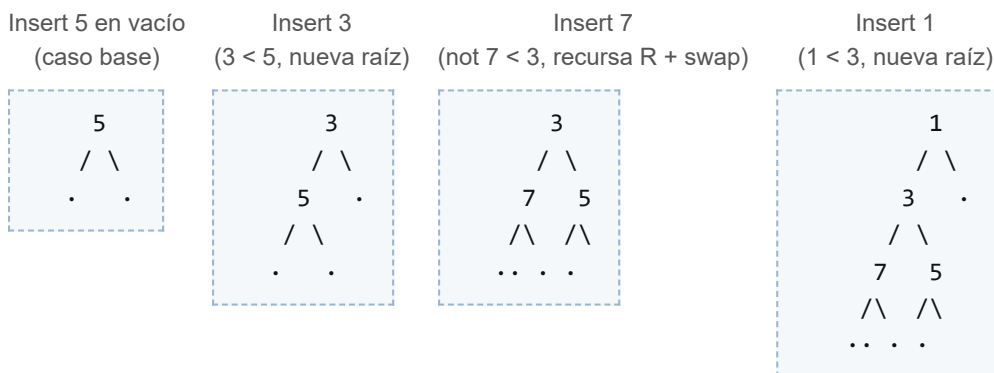
2. Estructura de los axiomas por operación

Operación	Patrón / condición	Resultado
Heap_Insert	insertar en heap vacío	BTree_Create(·, ·, i) as Heap_H (singleton)

	$i < x$ — el nuevo es estrictamente menor que la raíz	i sube a la raíz, el heap viejo entero queda como subárbol izquierdo, el derecho queda vacío
	$\text{not } (i < x)$	la raíz x permanece; i se inserta recursivamente en R ; swap de subárboles
Heap_Min	sobre BTree_Create(L, R, x) as Heap_H	x (la raíz)
Heap_DeleteMin	sobre BTree_Create(L, R, x) as Heap_H	Heap_Merge(L, R)
Heap_Merge	Heap_Merge(Heap_Empty, h)	h
	Heap_Merge(h , Heap_Empty)	h
	$x_1 < x_2$ — raíz del primero es menor	BTree_Create(Heap_Merge(R_1, h_2), L_1, x_1) as Heap_H (x_1 arriba, swap)
	$\text{not } (x_1 < x_2)$	BTree_Create(Heap_Merge(h_1, R_2), L_2, x_2) as Heap_H (x_2 arriba, swap)

3. Ejemplo: inserción sucesiva 5, 3, 7, 1

Trazando los axiomas de Heap_Insert paso a paso:



En cada paso la raíz es el mínimo. El paso "Insert 7" muestra el efecto del *skew*: como $\text{not } (7 < 3)$, recursamos en R (que era vacío, así que 7 va como singleton) y luego intercambiamos: el R nuevo (con 7) pasa a la izquierda, y la L original (que tenía 5) pasa a la derecha.

4. Garantías

Propiedad de heap (invariante): en todo nodo BTree_Create(L, R, x) as Heap_H, x es menor o igual que todos los elementos de L y R . Los axiomas la preservan:

- Caso base: singleton, trivialmente cumple.
- Heap_Insert caso $i < x$: i es menor que x (raíz vieja) y por inducción menor que todo descendiente del heap viejo $\rightarrow i$ es mínimo global.
- Heap_Insert caso $\text{not } (i < x)$: x sigue siendo menor o igual que el heap viejo (por hipótesis), e i ahora vive dentro de Heap_Insert(R, i), donde por inducción se preserva la propiedad.

- `Heap_Merge`: análogo, el menor de las dos raíces sube; los subárboles fusionados preservan la propiedad por inducción.

Encapsulamiento (invariante de API): tras `HIDE BTree_Create`, `Heap_Merge`, todo término `Heap_H` construible por un cliente tiene la forma `Heap_Insert(... Heap_Insert(Heap_Empty(), x_n) ..., x_1)`. Internamente esos términos se reescriben a `BTree_Create` vía los axiomas, y mantienen la propiedad de heap. **Es imposible** que un cliente produzca un `BTree_Create` mal-formado (por ejemplo con la raíz mayor que un descendiente).

Spec validada con el analyzer del workspace: 0 errores, 3 warnings (V1 sobre operaciones derivadas que reescriben constructores — falsos positivos esperados según la convención del proyecto).