

Heap_H — Min-heap clásico (completo por niveles)

Subtipo de *BTree* que preserva propiedad de heap y propiedad de forma. Constructores públicos: *Heap_Empty*, *Heap_Insert*. *BTree_Create* y 8 auxiliares ocultas.

Diferencia con *Heap_H.nereus (skew)*: aquella spec solo preservaba la propiedad de heap (raíz $\leq$ descendientes), pero no la forma — bastaba un insert con $i < \text{raíz}$ para producir un árbol "peinado" hacia la izquierda. Esta versión también preserva la forma clásica (árbol binario completo por niveles, último nivel llenado de izquierda a derecha), al costo de varias operaciones auxiliares para localizar la próxima posición de inserción y el último elemento en orden BFS.

Convención del workspace: los literales numéricos (0, 1) se tipan *Integer* en *Nereus*, no *Nat*; la spec usa *Heap_Size*: *Heap_H* $\rightarrow$ *Integer* consistentemente con *size*: *Collection* $\rightarrow$ *Integer* en la referencia del lenguaje. La regla " $|L| = |R|$ " se reescribe como " $\text{not } (|L| > |R|)$ " (en un árbol completo siempre $|L| \geq |R|$) para evitar que el parser confunda el = de la condición con el = del axioma.

1. Especificación

```
-- =====  
-- Heap_H — Min-heap CLÁSICO como subtipo de BTree.  
--  
-- Mantiene ambas propiedades:  
--   * Propiedad de heap:  la raíz es  $\leq$  que todos los descendientes.  
--   * Propiedad de forma: árbol binario completo por niveles,  
--                           último nivel llenado de izquierda a derecha.  
-- =====
```

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

OPERATIONS

```
-- — API pública —————
```

```
Heap_Empty:  $\rightarrow$  Heap_H;  
Heap_Insert: Heap_H(h) * Element(e)  $\rightarrow$  Heap_H;
```

```
Heap_Size:      Heap_H  $\rightarrow$  Integer;  
Heap_IsPerfect: Heap_H  $\rightarrow$  Boolean;
```

```
Heap_Min: Heap_H(h)  $\rightarrow$  Element  
  pre: not BTree_IsEmpty(h);  
Heap_DeleteMin: Heap_H(h)  $\rightarrow$  Heap_H  
  pre: not BTree_IsEmpty(h);
```

```
-- — Auxiliares internas (ocultas vía HIDE) —————
```

```
Heap_CombineL: Heap_H * Heap_H * Element  $\rightarrow$  Heap_H;  
Heap_CombineR: Heap_H * Heap_H * Element  $\rightarrow$  Heap_H;  
Heap_Last:      Heap_H(h)  $\rightarrow$  Element pre: not BTree_IsEmpty(h);  
Heap_RemoveLast: Heap_H(h)  $\rightarrow$  Heap_H pre: not BTree_IsEmpty(h);
```

```

Heap_ReplaceRoot: Heap_H(h) * Element -> Heap_H pre: not BTree_IsEmpty(h);
Heap_SiftDown: Heap_H -> Heap_H;
Heap_SDLeft: Heap_H * Heap_H * Element -> Heap_H;
Heap_SDBoth: Heap_H * Heap_H * Element -> Heap_H;

```

AXIOMS L, R, LL, LR, RL, RR: Heap_H; x, i, m, m1, m2: Element;

-- Heap_Size

```

Heap_Size(Heap_Empty() as Heap_H) = 0;
Heap_Size(BTree_Create(L, R, x) as Heap_H) =
  1 + Heap_Size(L as Heap_H) + Heap_Size(R as Heap_H);

```

-- Heap_IsPerfect

```

Heap_IsPerfect(Heap_Empty() as Heap_H) = True;
Heap_IsPerfect(BTree_Create(L, R, x) as Heap_H) =
  Heap_IsPerfect(L as Heap_H)
  and Heap_IsPerfect(R as Heap_H)
  and (Heap_Size(L as Heap_H) = Heap_Size(R as Heap_H));

```

-- Heap_Min, Heap_ReplaceRoot

```

Heap_Min(BTree_Create(L, R, x) as Heap_H) = x;
Heap_ReplaceRoot(BTree_Create(L, R, x) as Heap_H, m) =
  BTree_Create(L, R, m) as Heap_H;

```

-- Heap_Insert: 1 caso base + 3 reglas.

```

Heap_Insert(Heap_Empty() as Heap_H, i) =
  BTree_Create(Heap_Empty() as Heap_H,
    Heap_Empty() as Heap_H,
    i) as Heap_H;

```

-- Regla A: $|L| = |R| \rightarrow \text{izq.}$ (escrito como $\text{not}(|L| > |R|)$.)

```

not (Heap_Size(L as Heap_H) > Heap_Size(R as Heap_H)) implies
Heap_Insert(BTree_Create(L, R, x) as Heap_H, i) =
  Heap_CombineL(Heap_Insert(L as Heap_H, i), R as Heap_H, x);

```

-- Regla B: $|L| > |R| \wedge \neg \text{perfect}(L) \rightarrow \text{izq.}$

```

(Heap_Size(L as Heap_H) > Heap_Size(R as Heap_H)
  and not Heap_IsPerfect(L as Heap_H)) implies
Heap_Insert(BTree_Create(L, R, x) as Heap_H, i) =
  Heap_CombineL(Heap_Insert(L as Heap_H, i), R as Heap_H, x);

```

-- Regla C: $|L| > |R| \wedge \text{perfect}(L) \rightarrow \text{der.}$

```

(Heap_Size(L as Heap_H) > Heap_Size(R as Heap_H)
  and Heap_IsPerfect(L as Heap_H)) implies
Heap_Insert(BTree_Create(L, R, x) as Heap_H, i) =
  Heap_CombineR(L as Heap_H, Heap_Insert(R as Heap_H, i), x);

```

-- Heap_CombineL / Heap_CombineR: sift-up de un nivel.

```

(m < x) implies
Heap_CombineL(BTree_Create(LL, LR, m) as Heap_H, R, x) =
  BTree_Create(BTree_Create(LL, LR, x) as Heap_H, R, m) as Heap_H;
not (m < x) implies
Heap_CombineL(BTree_Create(LL, LR, m) as Heap_H, R, x) =
  BTree_Create(BTree_Create(LL, LR, m) as Heap_H, R, x) as Heap_H;
(m < x) implies
Heap_CombineR(L, BTree_Create(RL, RR, m) as Heap_H, x) =
  BTree_Create(L, BTree_Create(RL, RR, x) as Heap_H, m) as Heap_H;
not (m < x) implies
Heap_CombineR(L, BTree_Create(RL, RR, m) as Heap_H, x) =
  BTree_Create(L, BTree_Create(RL, RR, m) as Heap_H, x) as Heap_H;

```

```

-- Heap_Last: valor en la última posición BFS izq→der.
(BTree_IsEmpty(L) and BTree_IsEmpty(R)) implies
Heap_Last(BTree_Create(L, R, x) as Heap_H) = x;

(Heap_Size(L as Heap_H) > Heap_Size(R as Heap_H)
 and (not Heap_IsPerfect(L as Heap_H) or Heap_IsPerfect(R as Heap_H))) implies
Heap_Last(BTree_Create(L, R, x) as Heap_H) = Heap_Last(L as Heap_H);

(not (BTree_IsEmpty(L) and BTree_IsEmpty(R))
 and not (Heap_Size(L as Heap_H) > Heap_Size(R as Heap_H)
 and (not Heap_IsPerfect(L as Heap_H) or Heap_IsPerfect(R as Heap_H)))) implies
Heap_Last(BTree_Create(L, R, x) as Heap_H) = Heap_Last(R as Heap_H);

-- Heap_RemoveLast: mismo análisis, devuelve árbol sin último.
(BTree_IsEmpty(L) and BTree_IsEmpty(R)) implies
Heap_RemoveLast(BTree_Create(L, R, x) as Heap_H) = Heap_Empty() as Heap_H;

(Heap_Size(L as Heap_H) > Heap_Size(R as Heap_H)
 and (not Heap_IsPerfect(L as Heap_H) or Heap_IsPerfect(R as Heap_H))) implies
Heap_RemoveLast(BTree_Create(L, R, x) as Heap_H) =
  BTree_Create(Heap_RemoveLast(L as Heap_H), R, x) as Heap_H;

(not (BTree_IsEmpty(L) and BTree_IsEmpty(R))
 and not (Heap_Size(L as Heap_H) > Heap_Size(R as Heap_H)
 and (not Heap_IsPerfect(L as Heap_H) or Heap_IsPerfect(R as Heap_H)))) implies
Heap_RemoveLast(BTree_Create(L, R, x) as Heap_H) =
  BTree_Create(L, Heap_RemoveLast(R as Heap_H), x) as Heap_H;

-- Heap_DeleteMin
(BTree_IsEmpty(L) and BTree_IsEmpty(R)) implies
Heap_DeleteMin(BTree_Create(L, R, x) as Heap_H) = Heap_Empty() as Heap_H;

not (BTree_IsEmpty(L) and BTree_IsEmpty(R)) implies
Heap_DeleteMin(BTree_Create(L, R, x) as Heap_H) =
  Heap_SiftDown(
    Heap_ReplaceRoot(
      Heap_RemoveLast(BTree_Create(L, R, x) as Heap_H),
      Heap_Last(BTree_Create(L, R, x) as Heap_H)
    )
  );

-- Heap_SiftDown + helpers SDLeft/SDBoth (despachan según hijos).
(BTree_IsEmpty(L) and BTree_IsEmpty(R)) implies
Heap_SiftDown(BTree_Create(L, R, x) as Heap_H) = BTree_Create(L, R, x) as Heap_H;

(not BTree_IsEmpty(L) and BTree_IsEmpty(R)) implies
Heap_SiftDown(BTree_Create(L, R, x) as Heap_H) =
  Heap_SDLeft(L as Heap_H, R as Heap_H, x);

(not BTree_IsEmpty(L) and not BTree_IsEmpty(R)) implies
Heap_SiftDown(BTree_Create(L, R, x) as Heap_H) =
  Heap_SDBoth(L as Heap_H, R as Heap_H, x);

(m < x) implies
Heap_SDLeft(BTree_Create(LL, LR, m) as Heap_H, R, x) =
  BTree_Create(Heap_SiftDown(BTree_Create(LL, LR, x) as Heap_H), R, m) as Heap_H;
not (m < x) implies
Heap_SDLeft(BTree_Create(LL, LR, m) as Heap_H, R, x) =
  BTree_Create(BTree_Create(LL, LR, m) as Heap_H, R, x) as Heap_H;

```

```

-- SDBoth: 3 casos según relación de x con las dos raíces.
(not (m1 < x) and not (m2 < x)) implies
Heap_SDBoth(BTree_Create(LL, LR, m1) as Heap_H,
             BTree_Create(RL, RR, m2) as Heap_H, x) =
             BTree_Create(BTree_Create(LL, LR, m1) as Heap_H,
                           BTree_Create(RL, RR, m2) as Heap_H, x) as Heap_H;

((m1 < x) and not (m2 < m1)) implies
Heap_SDBoth(BTree_Create(LL, LR, m1) as Heap_H,
             BTree_Create(RL, RR, m2) as Heap_H, x) =
             BTree_Create(Heap_SiftDown(BTree_Create(LL, LR, x) as Heap_H),
                           BTree_Create(RL, RR, m2) as Heap_H, m1) as Heap_H;

((m2 < x) and (m2 < m1)) implies
Heap_SDBoth(BTree_Create(LL, LR, m1) as Heap_H,
             BTree_Create(RL, RR, m2) as Heap_H, x) =
             BTree_Create(BTree_Create(LL, LR, m1) as Heap_H,
                           Heap_SiftDown(BTree_Create(RL, RR, x) as Heap_H), m2) as Heap_H;

HIDE BTree_Create,
      Heap_CombineL, Heap_CombineR,
      Heap_Last, Heap_RemoveLast, Heap_ReplaceRoot,
      Heap_SiftDown, Heap_SDLleft, Heap_SDBoth

```

END-CLASS

2. Comparativa skew vs. clásico

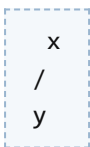
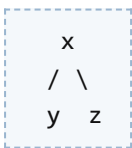
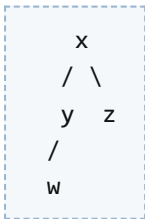
	Heap_H.nereus (skew)	Heap_H_complete.nereus (clásico)
Propiedad de heap	✓	✓
Propiedad de forma	X	✓
API pública	Empty, Insert, Min, DeleteMin	+ Size, IsPerfect
Aux. ocultas	1 (Merge)	8 (CombineL/R, Last, RemoveLast, ReplaceRoot, SiftDown, SDLleft, SDBoth)
Complejidad	O(log n) amortizado	O(log n) garantizado
Algoritmo Insert	merge con singleton + swap subárboles	localizar próxima posición BFS + sift-up por nivel
Algoritmo DeleteMin	merge de L y R	reemplazar raíz por último BFS + sift-down

3. Decisiones estructurales

Pregunta	Condición (sobre BTree_Create(L, R, x))	Acción
¿Dónde inserta el próximo elemento?	$ L = R $	en L (estrenar nivel nuevo)
	$ L > R \wedge \neg \text{perfect}(L)$	en L (continuar llenando L)
	$ L > R \wedge \text{perfect}(L)$	en R (L lleno, llenar R)
¿Dónde está el "último" elemento?	$ L > R \wedge (\neg \text{perfect}(L) \vee \text{perfect}(R))$	en L
	en otro caso (no-singleton)	en R
¿Es perfecto?	Heap_Empty sí; BTree_Create(L, R, x) sii $\text{perfect}(L) \wedge \text{perfect}(R) \wedge L = R $	

4. Cobertura de tamaños chicos

Para verificar las reglas, listo los árboles canónicos hasta $n = 7$ (todos válidos como heap completo); el campo "última posición" sirve para verificar la regla de Heap_Last.

n	Forma	(L , R)	Última posición
1		(0, 0)	raíz
2		(1, 0)	L
3		(1, 1)	R
4		(2, 1)	L

5	<pre> x / \ y z /\ w v </pre>	(3, 1)	L
6	<pre> x / \ y z /\ / w v u </pre>	(3, 2)	R
7	<pre> x / \ y z /\ /\ w v u t </pre>	(3, 3)	R

Spec validada con el analyzer del workspace: 0 errores, 13 warnings (V1 sobre operaciones derivadas que reescriben constructores — falsos positivos esperados).