

Traza — Heap_H clásico

Construcción con 6 Inserts (15, 12, 3, 7, 9, 11) seguida de DeleteMin. Ejercita las tres reglas de Insert con anidamiento profundo y un DeleteMin con SiftDown de dos niveles.

Término a evaluar:

```
Heap_DeleteMin(  
  Heap_Insert(  
    Heap_Insert(  
      Heap_Insert(  
        Heap_Insert(  
          Heap_Insert(  
            Heap_Insert(Heap_Empty() as Heap_H, 15),  
            12),  
          3),  
        7),  
      9),  
    11))
```

Spec: Heap_H_complete.nereus. Convención: · = Heap_Empty() as Heap_H.

Fase 1 — Construcción del heap (6 Inserts)

A1 — Insert 15 (caso base)

Sobre Heap_Empty → singleton.

```
BTree_Create(·, ·, 15) as Heap_H
```

tras Insert 15 — size 1

15

A2 — Insert 12 (Regla A: |L|=|R|=0)

Regla A (descender por L). Recursivo: Heap_Insert(·, 12) = BTree_Create(·, ·, 12).

Heap_CombineL(BTree_Create(·, ·, 12), ·, 15): m=12 < x=15 → swap.

```
BTree_Create(BTree_Create(·, ·, 15) as Heap_H, ·, 12) as Heap_H
```

tras Insert 12 — size 2

12
/
15

A3 — Insert 3 (Regla C: $|L|=1>|R|=0$, L perfecto)

Regla C (descender por R, porque L singleton ya está perfecto). Recursivo: `Heap_Insert(·, 3) = BTree_Create(·, ·, 3)`.

`Heap_CombineR(L, BTree_Create(·, ·, 3), 12): m=3 < x=12 → swap: 12 baja al subárbol derecho, 3 sube a la raíz.`

```
BTree_Create(  
  BTree_Create(·, ·, 15) as Heap_H,  
  BTree_Create(·, ·, 12) as Heap_H,  
  3) as Heap_H
```

tras Insert 3 — size 3

```
  3  
 / \  
15 12
```

A4 — Insert 7 (Regla A: $|L|=|R|=1$)

Regla A (descender por L). **Sub-recursión interna en `L=BTree_Create(·,·,15)`:**

- $|L'|=|R'|=0 \rightarrow$ Regla A. `Heap_Insert(·, 7) = BTree_Create(·, ·, 7)`.
- `Heap_CombineL(BTree_Create(·, ·, 7), ·, 15): 7 < 15 → swap`. Resultado interno:
`BTree_Create(BTree_Create(·,·,15), ·, 7)`.

Vuelta al outer: `Heap_CombineL(BTree_Create(BTree_Create(·,·,15), ·, 7), BTree_Create(·,·,12), 3): m=7 ≥ x=3 → no swap`.

```
BTree_Create(  
  BTree_Create(BTree_Create(·, ·, 15) as Heap_H, ·, 7) as Heap_H,  
  BTree_Create(·, ·, 12) as Heap_H,  
  3) as Heap_H
```

tras Insert 7 — size 4

```
    3  
   / \  
  7  12  
 /  
15
```

A5 — Insert 9 (Regla B: $|L|=2>|R|=1$, L no perfecto)

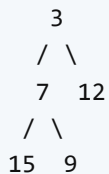
L tiene $|LL|=1$, $|LR|=0 \rightarrow$ no perfecto. Regla B (descender L para completarlo). **Sub-recursión en L:**

- $L = \text{BTree_Create}(\text{BTree_Create}(\cdot, \cdot, 15), \cdot, 7)$. $|L'|=1$, $|R'|=0$, L' singleton $\rightarrow$ perfecto. Regla C (descender R').
- $\text{Heap_Insert}(\cdot, 9) = \text{BTree_Create}(\cdot, \cdot, 9)$.
- $\text{Heap_CombineR}(\text{BTree_Create}(\cdot, \cdot, 15), \text{BTree_Create}(\cdot, \cdot, 9), 7): 9 \geq 7 \rightarrow$ no swap.
- Resultado interno: $\text{BTree_Create}(\text{BTree_Create}(\cdot, \cdot, 15), \text{BTree_Create}(\cdot, \cdot, 9), 7)$.

Vuelta al outer: $\text{Heap_CombineL}(\dots, \text{BTree_Create}(\cdot, \cdot, 12), 3): 7 \geq 3 \rightarrow$ no swap.

```
BTree_Create(  
  BTree_Create(BTree_Create( $\cdot$ ,  $\cdot$ , 15) as Heap_H,  
    BTree_Create( $\cdot$ ,  $\cdot$ , 9) as Heap_H, 7) as Heap_H,  
  BTree_Create( $\cdot$ ,  $\cdot$ , 12) as Heap_H,  
  3) as Heap_H
```

tras Insert 9 — size 5



A6 — Insert 11 (Regla C: $|L|=3>|R|=1$, L perfecto)

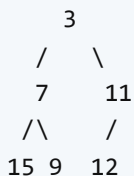
Ahora L tiene $|LL|=|LR|=1$, ambos singletons, $\rightarrow$ perfecto (3 nodos: árbol completo de altura 2). Regla C (descender R). **Sub-recursión en R= $\text{BTree_Create}(\cdot, \cdot, 12)$:**

- $|L'|=|R'|=0 \rightarrow$ Regla A. $\text{Heap_Insert}(\cdot, 11) = \text{BTree_Create}(\cdot, \cdot, 11)$.
- $\text{Heap_CombineL}(\text{BTree_Create}(\cdot, \cdot, 11), \cdot, 12): 11 < 12 \rightarrow$ swap. Resultado:
 $\text{BTree_Create}(\text{BTree_Create}(\cdot, \cdot, 12), \cdot, 11)$.

Vuelta al outer: $\text{Heap_CombineR}(L_{\text{perfecto}}, \text{BTree_Create}(\text{BTree_Create}(\cdot, \cdot, 12), \cdot, 11), 3): m=11 \geq x=3 \rightarrow$ no swap.

```
BTree_Create(  
  BTree_Create(BTree_Create( $\cdot$ ,  $\cdot$ , 15) as Heap_H,  
    BTree_Create( $\cdot$ ,  $\cdot$ , 9) as Heap_H, 7) as Heap_H,  
  BTree_Create(BTree_Create( $\cdot$ ,  $\cdot$ , 12) as Heap_H,  $\cdot$ , 11) as Heap_H,  
  3) as Heap_H
```

tras Insert 11 — size 6, heap construido



Posiciones BFS: 1=3, 2=7, 3=11, 4=15, 5=9, 6=12. Forma OK (completo niveles 1–2, último nivel parcial llenado izquierda $\rightarrow$ derecha hasta posición 6). Heap order OK: $3 \leq 7$, $3 \leq 11$, $7 \leq 15$, $7 \leq 9$, $11 \leq 12$.

Fase 2 — Heap_DeleteMin sobre el heap construido

Llamemos T al heap construido ($\text{BTree_Create}(L_T, R_T, 3)$ as Heap_H). Como T no es singleton, se aplica el axioma general:

```
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)
    )
  );
```

D1 — Calcular Heap_Last(T)

$T: |L_T|=3, |R_T|=2, |L_T| > |R_T| \checkmark$.

$\text{perfect}(L_T)? L_T$ tiene $|LL|=|LR|=1$, ambos perfectos $\rightarrow$ **sí**.

$\text{perfect}(R_T)? R_T = \text{BTree_Create}(\text{BTree_Create}(\cdot, \cdot, 12), \cdot, 11)$ con $|R_T L|=1, |R_T R|=0 \rightarrow$ **no**.

Condición "último en L": $|L| > |R| \wedge (\neg \text{perfect}(L) \vee \text{perfect}(R)) = T \wedge (F \vee F) = F \rightarrow$ último **en R**.

Recursión en $R_T = \text{BTree_Create}(\text{BTree_Create}(\cdot, \cdot, 12), \cdot, 11)$:

- $|L'|=1, |R'|=0, |L'| > |R'| \checkmark, \text{perfect}(L') \text{ singleton } \checkmark, \text{perfect}(R') \text{ empty } \checkmark$.
- Condición: $T \wedge (F \vee T) = T \rightarrow$ último en L' .
- $\text{Heap_Last}(\text{BTree_Create}(\cdot, \cdot, 12))$: singleton $\rightarrow$ **12**.

Resultado: $\text{Heap_Last}(T) =$ **12**.

D2 — Calcular Heap_RemoveLast(T)

Mismo análisis: último en R_T . Aplico axioma:

$\text{Heap_RemoveLast}(T) = \text{BTree_Create}(L_T, \text{Heap_RemoveLast}(R_T), 3)$.

Recursión: $\text{Heap_RemoveLast}(R_T)$ con último en L' $\rightarrow$

$= \text{BTree_Create}(\text{Heap_RemoveLast}(\text{BTree_Create}(\cdot, \cdot, 12)), \cdot, 11)$

$= \text{BTree_Create}(\text{Heap_Empty}, \cdot, 11)$ (caso singleton)

$= \text{BTree_Create}(\cdot, \cdot, 11)$ **as Heap_H.**

```
BTree_Create(  
  BTree_Create(BTree_Create( $\cdot$ ,  $\cdot$ , 15) as Heap_H,  
    BTree_Create( $\cdot$ ,  $\cdot$ , 9) as Heap_H, 7) as Heap_H,  
  BTree_Create( $\cdot$ ,  $\cdot$ , 11) as Heap_H,  
  3) as Heap_H
```

T sin la última posición (raíz aún 3) — size 5

```
      3  
     /\   
    7  11  
   /\   
  15 9
```

D3 — Heap_ReplaceRoot(..., 12)

Axioma: $\text{Heap_ReplaceRoot}(\text{BTree_Create}(L, R, x), m) = \text{BTree_Create}(L, R, m)$. Reemplazo 3 por 12:

```
BTree_Create(  
  BTree_Create(BTree_Create( $\cdot$ ,  $\cdot$ , 15) as Heap_H,  
    BTree_Create( $\cdot$ ,  $\cdot$ , 9) as Heap_H, 7) as Heap_H,  
  BTree_Create( $\cdot$ ,  $\cdot$ , 11) as Heap_H,  
  12) as Heap_H
```

Antes de SiftDown (la raíz 12 viola la propiedad heap)

```
      12  
     /\   
    7  11  
   /\   
  15 9
```

12 está por encima de 7 (hijo izq) y de 11 (hijo der). El SiftDown lo baja al lugar correcto.

D4 — Heap_SiftDown: nivel 1 (SDBoth, caso b)

Ambos hijos no vacíos $\rightarrow$ Heap_SDBoth(L, R, 12) con $m1=7$ (raíz de L), $m2=11$ (raíz de R), $x=12$.

Caso	Condición	Aplica
(a)	$\neg(7 < 12) \wedge \neg(11 < 12) = F \wedge F$	No
(b)	$(7 < 12) \wedge \neg(11 < 7) = T \wedge T$	Sí $\leftarrow$
(c)	$(11 < 12) \wedge (11 < 7) = T \wedge F$	No

Caso (b): el menor hijo es $m1=7$, swap con izquierda y recurrir.

RHS: BTree_Create(Heap_SiftDown(BTree_Create(LL, LR, 12)), R, 7) con $LL=BTree_Create(\cdot, \cdot, 15)$, $LR=BTree_Create(\cdot, \cdot, 9)$.

Sub-término pendiente de SiftDown: BTree_Create(BTree_Create($\cdot, \cdot, 15$), BTree_Create($\cdot, \cdot, 9$), 12).

D5 — Heap_SiftDown recursivo: nivel 2 (SDBoth, caso c)

Sub-árbol: BTree_Create(BTree_Create($\cdot, \cdot, 15$), BTree_Create($\cdot, \cdot, 9$), 12). Ambos hijos no vacíos $\rightarrow$ SDBoth con $m1=15$, $m2=9$, $x=12$.

Caso	Condición	Aplica
(a)	$\neg(15 < 12) \wedge \neg(9 < 12) = T \wedge F$	No
(b)	$(15 < 12) \wedge \neg(9 < 15) = F \wedge T$	No
(c)	$(9 < 12) \wedge (9 < 15) = T \wedge T$	Sí $\leftarrow$

Caso (c): el menor hijo es $m2=9$ (estrictamente menor que $m1=15$), swap con derecha y recurrir.

RHS: BTree_Create(BTree_Create($\cdot, \cdot, 15$), Heap_SiftDown(BTree_Create($\cdot, \cdot, 12$)), 9).

Sub-término pendiente: Heap_SiftDown(BTree_Create($\cdot, \cdot, 12$)) — hoja, no hace nada $\rightarrow$ BTree_Create($\cdot, \cdot, 12$).

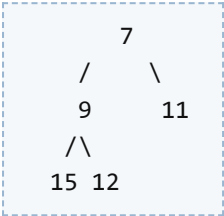
```
BTree_Create(  
  BTree_Create( $\cdot, \cdot, 15$ ) as Heap_H,  
  BTree_Create( $\cdot, \cdot, 12$ ) as Heap_H,  
  9) as Heap_H
```

Sub-árbol post-sift: 9 a la raíz, 15 a la izquierda, 12 a la derecha. Propiedad heap localmente OK: $9 \leq 15$, $9 \leq 12$.

D6 — Sustituir resultado del sift recursivo en el árbol del paso D4

```
BTree_Create(  
  BTree_Create(BTree_Create(., ., 15) as Heap_H,  
    BTree_Create(., ., 12) as Heap_H, 9) as Heap_H,  
  BTree_Create(., ., 11) as Heap_H,  
  7) as Heap_H
```

Forma normal del DeleteMin — size 5



Verificación y resumen

Aspecto	Antes del DeleteMin	Después del DeleteMin
Elementos	{3, 7, 9, 11, 12, 15}	{7, 9, 11, 12, 15}
Mínimo	3	7
Tamaño	6	5
Posiciones BFS	1=3, 2=7, 3=11, 4=15, 5=9, 6=12	1=7, 2=9, 3=11, 4=15, 5=12
Forma (completo + último izq→der)	✓	✓
Heap order	$3 \leq 7, 11$; $7 \leq 15, 9$; $11 \leq 12$ ✓	$7 \leq 9, 11$; $9 \leq 15, 12$ ✓

Recorrido del "último" 12: al iniciar el DeleteMin, 12 estaba en la posición 6 (hoja izquierda del subárbol derecho). El proceso fue:

- 1. **Detección (D1):** Heap_Last recorrió por R_T (porque L_T ya era perfecto), luego por L' (porque ahí estaba el último BFS), y encontró 12.
- 2. **Promoción a raíz (D3):** 12 subió temporalmente a la raíz, violando el heap order.
- 3. **Descenso por sift (D4–D6):** bajó por la izquierda (porque 7 era el menor hijo y $7 < 12$), llegando al subárbol con raíz vieja 7. Después, dentro de ese subárbol, bajó por la derecha (porque 9 era estrictamente menor que 15) y quedó como hoja derecha de 9.

Resultado: 12 terminó dos niveles abajo de donde empezó, atravesando el árbol diagonalmente. Eso es lo que hace el sift-down clásico.

Comparación con el ejemplo anterior (Insert 5, 3, 8, 1 + DeleteMin): en aquél el SiftDown se resolvió en un solo nivel (D4: caso b, recursión sobre hoja). Acá el SiftDown atravesó dos niveles (D4 caso b → D5 caso c) porque el último elemento (12) era mayor que dos descendientes seguidos. Eso ilustra que el costo de SiftDown crece con la altura del árbol — $O(\log n)$ en el peor caso.

Traza generada para la spec `Heap_H_complete.nereus`. Pasos: 6 Inserts (con anidamiento de hasta 2 niveles) + 6 sub-pasos del DeleteMin (Last + RemoveLast + ReplaceRoot + SiftDown de 2 niveles + hoja terminal).