```
*******************************************************************************
************                     Interfaz                          ************
*******************************************************************************

Parametros Formales
    géneros α
    función • < • (in a1 : α, a2 : α) → res : bool
        Pre ≡ {true}
        Post ≡ {res =obs (a1 < a2)}
        Complejidad: O(lower(a1, a2))
        Descripción: funcion de comparación de αs

Se explica con: ColaDePrioridadExtendida(α), IteradorUnidireccional(α)
Generos: ColaPrioridad(α), itCola(α)

*******************************************************************************
************                    Operaciones                        ************
*******************************************************************************

Crear() → res : ColaPrioridad(α)
Pre ≡ {true}
Post ≡ {res =obs vacia}
Complejidad: O(1)
Descripción: Crea una cola vacia.

Encolar(in/out t : ColaPrioridad(α), in e : α) → res : itCola(α)
Pre ≡ {¬(e ∈ t) ∧ t_0 =obs t}
Post ≡ {t =obs encolar(e, t_0) ∧ alias(Actual(res) =obs e)}
Complejidad: O(log(Tamaño(t)))
Descripción: Inserta un elemento en la cola y devuelve un iterador posicionado en el elemento
agregado.
Aliasing: El iterador se invalida sii se borra el elemento utizando Desencolar o Borrar

Desencolar(in/out t : ColaPrioridad(α)) → res : α
Pre ≡ {¬vacia?(t) ∧ t =obs t_0}
Post ≡ {t =obs desencolar(t_0) ∧ res =obs proximo(t_0)}
Complejidad: O(log(Tamaño(t)))
Descripción: Desencola el elemento con mayor prioridad.

Tamaño(in t : ColaPrioridad(α)) → res : nat
Pre ≡ {true}
Post ≡ {res =obs #t}
Complejidad: O(1)
Descripción: Devuelve la cantidad de elementos en la cola.

Borrar(in/out t : ColaPrioridad(α), in i : itCola(α)) → res : α
Pre ≡ {t_0 =obs t}
Post ≡ {t =obs borrar(Actual(i), t_0)}
Complejidad: O(log(Tamaño(t)))
Descripción: Borra el elemento al que apunta el iterador.
Aliasing: Invalida el iterador.

Copiar(in t: ColaPrioridad(α)) → res : ColaPrioridad(α)
Pre ≡ {true}
Post ≡ {t =obs res}
Complejidad: O(#t)
Descripcion: Realiza una copia de la cola de prioridad.


TAD ColaDePrioridadExtendida(α)
    extiende colaPrior(α)
    otras operaciones (exportadas)
        #• : colaPrior(α) → nat
        • ∈ • : α × colaPrior(α) → bool
        borrar : α e × colaPrior(α) c → colaPrior(α) {e ∈ c}
    axiomas
        #c ≡ if vacia?(c) then 0 else 1 + #desencolar(c) fi
        x ∈ c ≡ ¬vacia?(c) ∧_L (x =obs proximo(c) ∨_L x ∈ desencolar(c))
        borrar(e, c) ≡
```

```
            if e =obs proximo(c) then
                if e ∈ desencolar(c) then borrar(e, desencolar(c))
                else desencolar(c) fi
            else encolar(proximo(c), borrar(e, desencolar(c))) fi
Fin TAD

********************************************************************************
************                    Representación                    ************
********************************************************************************

ColaPrioridad(α) se representa con estr(α)
    donde estr(α) es tupla(
                        cabeza : puntero(nodo(α)),
                        ultimo : puntero(nodo(α)),
                        tamaño : nat)
    donde nodo(α) es tupla(
                        arr    : puntero(nodo(α)),
                        izq    : puntero(nodo(α)),
                        der    : puntero(nodo(α)),
                        dato   : α)

Rep: ^(estr(α)) → boolean
Rep(e) ≡ true ⟺
    (e.cabeza =obs NULL ⟺ e.ultimo =obs NULL) ∧
    e.tamaño =obs Tamaño(e.cabeza) ∧
    InvPadres(e.cab) ∧
    No hay ciclos en el arbol ∧ₗ
    MaxHeap(e.cab) ∧
    Balanceado(e.cab) ∧
    Izquierdista(e.cab) ∧ₗ
    (¬(e.cabeza =obs NULL) ⟹ₗ
        e.cabeza→arr =obs NULL ∧
        e.ultimo ES EL ULTIMO AGREGADO)

InvPadres: puntero(nodo(α)) → bool
InvPadres(p) ≡
    if p =obs NULL then true
    else
        (¬(p→izq =obs NULL) ⟹ₗ p→izq→arr =obs p) ∧
        (¬(p→der =obs NULL) ⟹ₗ p→der→arr =obs p) ∧
        InvPadres(p→izq) ∧
        InvPadres(p→der)
    fi

MaxHeap: puntero(nodo(α)) → bool
MaxHeap(p) ≡
    if p =obs NULL then true
    else
        (¬(p→izq =obs NULL) ⟹ₗ p→izq→dato < p→dato) ∧
        (¬(p→der =obs NULL) ⟹ₗ p→der→dato < p→dato) ∧
        MaxHeap(p→izq) ∧
        MaxHeap(p→der)
    fi

Balanceado: puntero(nodo(α)) → bool
Balanceado(p) ≡
    (p =obs NULL) ∨ₗ (|Altura(p→izq) - Altura(p→der)| ≤ 1) ∧
    Balanceado(p→izq) ∧ Balanceado(p→der)

Altura: puntero(nodo(α)) → nat
Altura(p) ≡ if p =obs NULL then 0 else 1 + max(Altura(p→izq), Altura(p→der)) fi

Izquierdista: puntero(nodo(α)) → bool
Izquierdista(p) ≡
    (p =obs NULL) ∨ₗ (¬(p→der =obs NULL) ⟹ₗ ¬(p→izq =obs NULL)) ∧
    Izquierdista(p→izq) ∧ Izquierdista(p→der)

Tamaño: puntero(nodo(α)) → nat
Tamaño(p) ≡ if p =obs NULL then 0 else 1 + Tamaño(p→izq) + Tamaño(p→der) fi
```

```
Abs: ^(estr(α)) e → ColaDePrioridadExtendida(α)                           {Rep(e)}
(∀e : ^(estr(α))) Abs(e) =obs t /
    vacia?(t) =obs (e.tam =obs 0) ∧
    ¬vacia?(t) ⟹ₗ
        proximo(t) =obs e.cab→dato ∧
        (∀e : α) (((e ∈ t) ∧ ¬(e =obs proximo(t))) ⟺ (e ∈ desencolar(t)))

********************************************************************************
************                      Algoritmos                        ************
********************************************************************************

iCrear() → res : estr(α)
    res ← ⟨cabeza: NULL, ultimo: NULL, tamaño: 0⟩
end function

iTamaño(in t : estr(α)) → res : nat
    res ← t.tamaño
end function

iEncolar(in/out t : estr(α), in e : α) → res : itCola(α)
    var tmp : puntero(nodo(α)) ← &⟨arr: NULL, izq: NULL, der: NULL, dato: e⟩

    if Tamaño(t) == 0 then
        tmp→arr ← NULL
        t.cabeza ← tmp
    else if Tamaño(t) == 1 then
        tmp→arr ← t.cabeza
        t.cabeza→izq ← tmp
    else
        if t.ultimo→arr→izq == t.ultimo then
            tmp→arr ← t.ultimo→arr
            t.ultimo→arr→der ← tmp
        else
            var cur : puntero(nodo(α)) ← t.ultimo

            while cur→arr != NULL ∧ cur→arr→izq != cur do
                cur ← cur→arr
            end while

            if cur→arr != NULL then
                cur ← cur→arr→der
            fi

            while cur→izq != NULL do
                cur ← cur→izq
            end while

            tmp→arr ← cur
            cur→izq ← tmp
        end if
    end if

    t.ultimo ← tmp
    t.tamaño++

    Subir(t.ultimo)
    res ← crearIter(t, p)
end function
```

En el peor caso, tenemos un árbol con más de 1 elemento, en el que el último nodo agregado está a la derecha de su padre, fundamentalmente el peor es cuando el árbol es completo.
En este caso, observemos que lo que sucederá es que subiremos hasta la raíz del árbol, y luego se bajaremos hacia el último nodo a la izquierda del árbol. Es decir, recorreremos dos veces la altura del árbol (2*log(#t)). Luego, en absolutamente todos los casos haremos a lo sumo log(#t) pasos para restablecer el invariante utilizando Subir. Es decir, haremos 3*log(#t), por lo que en el peor de los casos es O(log(#t)).

```
iDesencolar(in/out t : estr(α)) → res : α
    res ← Eliminar(t, t.cabeza)
end function

iBorrar(in/out t : estr(α), in i : itCola(α)) → res : α
    res ← Eliminar(t, i)
end function


iCopiar(in t : estr(α)) → res : estr(α)
    res ← ⟨cabeza: &iCopiarNodo(*t.cabeza), ultimo: NULL, tamaño: t.tamaño⟩
    var camino_ultimo : Pila(α)
    var tmp : Puntero(Nodo(α)) ← t.ultimo
    while tmp→arr != NULL
        Apilar(camino_ultimo, tmp→dato)
    end while

    tmp ← res.cabeza

    while ¬EsVacia?(camino_ultimo)
        if tmp→izq→dato == Tope(camino_ultimo)
            tmp ← tmp→izq
        else
            tmp ← tmp→der
        end if
        Desapilar(camino_ultimo)
    end while

    res.ultimo ← tmp
end function


Pre ≡ {true}
Post ≡ {res =obs n}
Complejidad: O(Cantidad de nodos subyacentes)
Descripcion: Devuelve una copia del nodo n.
iCopiarNodo(in n : nodo(α)) → res : nodo(α)
    res ← ⟨arr: NULL, izq: NULL, der: NULL, dato: e⟩

    if n.izq != NULL
        res.izq ← &iCopiarNodo(*n.izq)
        res.izq→arr ← &res
    end fi

    if n.der != NULL
        res.der ← &iCopiarNodo(*n.der)
        res.der→arr ← &res
    end fi

    res.e ← Copiar(e)

end function
```

```
Pre ≡ {¬(p =obs NULL) ∧ₗ p es un puntero de la estructura de datos ∧ t₀ =obs t}
Post ≡ {t =obs borrar(p→dato, t₀) ∧ res =obs p→dato}
Descripcion: Elimina el dato al que apunta el puntero.
Complejidad: O(log(Tamaño(t)))
iEliminar(in/out t : estr(α), in p : puntero(nodo(α))) → res : α
    res ← p→dato

    if Tamaño(t) == 1 then
        t.ultimo ← NULL
        t.cabeza ← NULL
    else
        p→dato ← t.ultimo→dato

        if t.ultimo→arr→izq == t.ultimo then
            var cur : puntero(nodo(α)) = t.ultimo

            while cur→arr != NULL ∧ cur→arr→der != cur do
                cur ← cur→arr
            end while

            if cur→arr != NULL then
                cur ← cur→arr→izq
            fi

            while cur→der != NULL do
                cur ← cur→der
            end while

            t.ultimo→arr→izq ← NULL
        else
            t.ultimo ← t.ultimo→arr→izq
            t.ultimo→arr→der ← NULL
        end if

        Bajar(p)
    end if

    t.tamaño--
end function
```

En el peor caso, tenemos un árbol con más de 1 elemento, en el que el último nodo agregado está a la izquierda de su padre, fundamentalmente el peor es cuando el árbol tiene un nodo de más para ser completo.
En este caso, observemos que lo que sucederá es que subiremos hasta la raíz del árbol, y luego se bajaremos hacia el último nodo a la derecha del árbol. Es decir, recorreremos dos veces la altura del árbol (2*log(#t)). Luego, en absolutamente todos los casos haremos a lo sumo log(#t) pasos para restablecer el invariante utilizando Bajar. Es decir, haremos 3*log(#t), por lo que en el peor de los casos es O(log(#t)).

```
Pre ≡ {¬(p =obs NULL)}
Post ≡ {Se restableció el rep de la estructura de datos}
Complejidad: O(log(#t))
Descripción: Restablece el invariante si el nodo al que apunta p está fuera del
lugar que le corresponde.
iSubir(in/out p : puntero(nodo(α)))
    while p→arr != NULL ∧ p→arr→dato < p→dato do
        var tmp : α ← p→arr→dato
        p→arr→dato ← p→dato
        p→dato ← tmp
        p ← p→arr
    end while
end function
```

Observemos que en el peor caso, se va a iterar hasta que p→arr sea NULL. Pero el invariante de representación asegura que el único caso en que esto sucede es que el nodo sea la raiz. Es decir, desde una posición arbitraria la máxima cantidad de pasos es la altura del árbol, por lo que en el peor caso, la complejidad de iSubir es log(#t).

```
Pre ≡ {¬(p =obs NULL)}
Post ≡ {Se restableció el rep de la estructura de datos}
Complejidad: O(log(#t))
Descripción: Restablece el invariante si el nodo al que apunta p está fuera del
lugar que le corresponde.
iBajar(in/out p : puntero(nodo(α)))
    while    (p→izq != NULL ∧ p→dato < p→izq→dato) ∨
            (p→der != NULL ∧ p→der→dato < p→dato) do
        if p→izq != NULL then
            var tmp : α ← p→izq→dato
            p→izq→dato ← p→dato
            p→dato ← tmp
        else
            if p→der != NULL then
                var tmp : α ← p→der→dato
                p→der→dato ← p→dato
                p→dato ← p→der→dato
            end if
        end if
    end while
end function
```

Observemos que en el peor caso, se va a iterar hasta que tanto p→izq como p→der. sean NULL. Pero el invariante de representación asegura que el único caso en que esto sucede es que el nodo al que se llega sea una hoja. Es decir, desde una posición arbitraria la máxima cantidad de pasos es la altura del árbol, por lo que en el peor caso, la complejidad de iBajar es log(#t).

```
********************************************************************************
***********                 Operaciones del iterador               ************
********************************************************************************
```

Ninguna.

```
********************************************************************************
***********               Representación del iterador              ************
********************************************************************************
```

```
itCola(α) se representa con it(α)
    donde it(α) es puntero(nodo(α))

Rep: ^(it(α)) → boolean
Rep(e) ≡ true ⟺ ¬(e =obs NULL)

Abs: ^(it(α)) → IteradorUnidireccional(α)
(∀e : ^(it(α))) Abs(e) =obs i / Siguientes(i) =obs *e • <>
```

```
********************************************************************************
***********                 Algorítmos del Iterador                ************
********************************************************************************
```

```
Pre ≡ {p es un puntero en la estructura de t}
Post ≡ {itCola es un iterador posicionado en p}
Complejidad: O(1)
Descripción: Crea un puntero disfrazado de iterador.
crearIter(in t : colaPrioridad(α), p : puntero(nodo(α))) → res : itCola(α)
    res ← p
end function
```