

```

*****
*****                                Interfaz                                *****
*****

```

Parametros Formales

```

géneros  $\alpha$ 
función • < • (in  $a_1 : \alpha$ ,  $a_2 : \alpha$ )  $\rightarrow$  res : bool
    Pre  $\equiv \{true\}$ 
    Post  $\equiv \{res = obs(a_1 < a_2)\}$ 
    Complejidad:  $O(lower(a_1, a_2))$ 
    Descripción: funcion de comparación de  $\alpha$ s

```

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

```

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

```

```

Crear()  $\rightarrow$  res : ColaPrioridad( $\alpha$ )
Pre  $\equiv \{true\}$ 
Post  $\equiv \{res = obs vacia\}$ 
Complejidad:  $O(1)$ 
Descripción: Crea una cola vacia.

```

```

Encolar(in/out t : ColaPrioridad( $\alpha$ ), in e :  $\alpha$ )  $\rightarrow$  res : itCola( $\alpha$ )
Pre  $\equiv \{\neg(e \in t) \wedge t_0 = obs t\}$ 
Post  $\equiv \{t = obs encolar(e, t_0) \wedge alias(Actual(res) = obs e)\}$ 
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( $\alpha$ ))  $\rightarrow$  res :  $\alpha$ 
Pre  $\equiv \{\neg vacia?(t) \wedge t = obs t_0\}$ 
Post  $\equiv \{t = obs desencolar(t_0) \wedge res = obs proximo(t_0)\}$ 
Complejidad:  $O(\log(Tamaño(t)))$ 
Descripción: Desencola el elemento con mayor prioridad.

```

```

Tamaño(in t : ColaPrioridad( $\alpha$ ))  $\rightarrow$  res : nat
Pre  $\equiv \{true\}$ 
Post  $\equiv \{res = obs \#t\}$ 
Complejidad:  $O(1)$ 
Descripción: Devuelve la cantidad de elementos en la cola.

```

```

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

```

```

Copiar(in t : ColaPrioridad( $\alpha$ ))  $\rightarrow$  res : ColaPrioridad( $\alpha$ )
Pre  $\equiv \{true\}$ 
Post  $\equiv \{t = obs res\}$ 
Complejidad:  $O(\#t)$ 
Descripcion: Realiza una copia de la cola de prioridad.

```

```

TAD ColaDePrioridadExtendida( $\alpha$ )
    extiende colaPrior( $\alpha$ )
    otras operaciones (exportadas)
        #• : colaPrior( $\alpha$ )  $\rightarrow$  nat
        •  $\in$  • :  $\alpha \times colaPrior(\alpha) \rightarrow bool$ 
        borrar :  $\alpha \times colaPrior(\alpha) \times c \rightarrow colaPrior(\alpha) \{e \in c\}$ 
    axiomas
        #c  $\equiv$  if vacia?(c) then 0 else 1 + #desencolar(c) fi
        x  $\in$  c  $\equiv$   $\neg vacia?(c) \wedge$  (x = obs proximo(c)  $\vee$  x  $\in$  desencolar(c))
        borrar(e, c)  $\equiv$ 

```

```
Tamaño(p) ≡ if p = obs NULL then 0 else 1 + Tamaño(p→izq) + Tamaño(p→der) fi
```

```

Abs:  $\wedge(\text{estr}(\alpha)) e \rightarrow \text{ColaDePrioridadExtendida}(\alpha)$  {Rep(e)}
( $\forall e : \wedge(\text{estr}(\alpha))$ ) Abs(e) = obs t /
  vacia?(t) = obs (e.tam = obs 0)  $\wedge$ 
   $\neg \text{vacia?}(t) \Rightarrow$ 
    proximo(t) = obs e.cab-dato  $\wedge$ 
    ( $\forall e : \alpha$ ) ((e  $\in$  t)  $\wedge$   $\neg$ (e = obs proximo(t)))  $\Leftrightarrow$  (e  $\in$  desencolar(t))

```

```

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

```

```

iCrear()  $\rightarrow$  res : estr( $\alpha$ )
  res  $\leftarrow$  {cabeza: NULL, ultimo: NULL, tamaño: 0}
end function

iTamaño(in t : estr( $\alpha$ ))  $\rightarrow$  res : nat
  res  $\leftarrow$  t.tamaño
end function

iEncolar(in/out t : estr( $\alpha$ ), in e :  $\alpha$ )  $\rightarrow$  res : itCola( $\alpha$ )
  var tmp : puntero(nodo( $\alpha$ ))  $\leftarrow$  &{arr: NULL, izq: NULL, der: NULL, dato: e}

  if Tamaño(t) == 0 then
    tmp $\rightarrow$ arr  $\leftarrow$  NULL
    t.cabeza  $\leftarrow$  tmp
  else if Tamaño(t) == 1 then
    tmp $\rightarrow$ arr  $\leftarrow$  t.cabeza
    t.cabeza $\rightarrow$ izq  $\leftarrow$  tmp
  else
    if t.ultimo $\rightarrow$ arr $\rightarrow$ izq == t.ultimo then
      tmp $\rightarrow$ arr  $\leftarrow$  t.ultimo $\rightarrow$ arr
      t.ultimo $\rightarrow$ arr $\rightarrow$ der  $\leftarrow$  tmp
    else
      var cur : puntero(nodo( $\alpha$ ))  $\leftarrow$  t.ultimo

      while cur $\rightarrow$ arr != NULL  $\wedge$  cur $\rightarrow$ arr $\rightarrow$ izq != cur do
        cur  $\leftarrow$  cur $\rightarrow$ arr
      end while

      if cur $\rightarrow$ arr != NULL then
        cur  $\leftarrow$  cur $\rightarrow$ arr $\rightarrow$ der
      fi

      while cur $\rightarrow$ izq != NULL do
        cur  $\leftarrow$  cur $\rightarrow$ izq
      end while

      tmp $\rightarrow$ arr  $\leftarrow$  cur
      cur $\rightarrow$ izq  $\leftarrow$  tmp
    end if
  end if

  t.ultimo  $\leftarrow$  tmp
  t.tamaño++

  Subir(t.ultimo)
  res  $\leftarrow$  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 ($2 \cdot \log(\#t)$). Luego, en absolutamente todos los casos haremos a lo sumo $\log(\#t)$ pasos para restablecer el invariante utilizando Subir. Es decir, haremos $3 \cdot \log(\#t)$, por lo que en el peor de los casos es $O(\log(\#t))$.

```

iDesencolar(in/out t : estr( $\alpha$ ))  $\rightarrow$  res :  $\alpha$ 
    res  $\leftarrow$  Eliminar(t, t.cabeza)
end function

```

```

iBorrar(in/out t : estr( $\alpha$ ), in i : itCola( $\alpha$ ))  $\rightarrow$  res :  $\alpha$ 
    res  $\leftarrow$  Eliminar(t, i)
end function

```

```

iCopiar(in t : estr( $\alpha$ ))  $\rightarrow$  res : estr( $\alpha$ )
    res  $\leftarrow$  {cabeza: &iCopiarNodo(*t.cabeza), ultimo: NULL, tamaño: t.tamaño}
    var camino_ultimo : Pila( $\alpha$ )
    var tmp : Puntero(Nodo( $\alpha$ ))  $\leftarrow$  t.ultimo
    while tmp $\rightarrow$ arr  $\neq$  NULL
        Apilar(camino_ultimo, tmp $\rightarrow$ dato)
        tmp  $\leftarrow$  tmp $\rightarrow$ arr
    end while

    tmp  $\leftarrow$  res.cabeza

    while  $\neg$ EsVacía?(camino_ultimo)
        if tmp $\rightarrow$ izq $\rightarrow$ dato == Tope(camino_ultimo)
            tmp  $\leftarrow$  tmp $\rightarrow$ izq
        else
            tmp  $\leftarrow$  tmp $\rightarrow$ der
        end if
        Desapilar(camino_ultimo)
    end while

    res.ultimo  $\leftarrow$  tmp
end function

```

```

Pre  $\equiv$  {true}
Post  $\equiv$  {res =obs n}
Complejidad: O(Cantidad de nodos subyacentes)
Descripción: Devuelve una copia del nodo n.
iCopiarNodo(in n : nodo( $\alpha$ ))  $\rightarrow$  res : nodo( $\alpha$ )
    res  $\leftarrow$  {arr: NULL, izq: NULL, der: NULL, dato: e}

    if n.izq  $\neq$  NULL
        res.izq  $\leftarrow$  &iCopiarNodo(*n.izq)
        res.izq $\rightarrow$ arr  $\leftarrow$  &res
    end fi

    if n.der  $\neq$  NULL
        res.der  $\leftarrow$  &iCopiarNodo(*n.der)
        res.der $\rightarrow$ arr  $\leftarrow$  &res
    end fi

    res.e  $\leftarrow$  Copiar(e)

end function

```

```

Pre  $\equiv \{ \neg(p = \text{obs NULL}) \wedge p \text{ es un puntero de la estructura de datos} \wedge t_0 = \text{obs } t \}$ 
Post  $\equiv \{ t = \text{obs borrar}(p \rightarrow \text{dato}, t_0) \wedge \text{res} = \text{obs } p \rightarrow \text{dato} \}$ 
Descripcion: Elimina el dato al que apunta el puntero.
Complejidad:  $O(\log(\text{Tamaño}(t)))$ 
iEliminar(in/out t : estr( $\alpha$ ), in p : puntero(nodo( $\alpha$ )))  $\rightarrow$  res :  $\alpha$ 
    res  $\leftarrow p \rightarrow \text{dato}$ 

    if Tamaño(t) == 1 then
        t.ultimo  $\leftarrow$  NULL
        t.cabeza  $\leftarrow$  NULL
    else
        p  $\rightarrow$  dato  $\leftarrow$  t.ultimo  $\rightarrow$  dato

        if t.ultimo  $\rightarrow$  arr  $\rightarrow$  izq == t.ultimo then
            var cur : puntero(nodo( $\alpha$ )) = t.ultimo

            while cur  $\rightarrow$  arr != NULL  $\wedge$  cur  $\rightarrow$  arr  $\rightarrow$  der != cur do
                cur  $\leftarrow$  cur  $\rightarrow$  arr
            end while

            if cur  $\rightarrow$  arr != NULL then
                cur  $\leftarrow$  cur  $\rightarrow$  arr  $\rightarrow$  izq
            fi

            while cur  $\rightarrow$  der != NULL do
                cur  $\leftarrow$  cur  $\rightarrow$  der
            end while

            t.ultimo  $\rightarrow$  arr  $\rightarrow$  izq  $\leftarrow$  NULL
        else
            t.ultimo  $\leftarrow$  t.ultimo  $\rightarrow$  arr  $\rightarrow$  izq
            t.ultimo  $\rightarrow$  arr  $\rightarrow$  der  $\leftarrow$  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  $\equiv \{ \neg(p = \text{obs NULL}) \}$ 
Post  $\equiv \{ \text{Se restableció el rep de la estructura de datos} \}$ 
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( $\alpha$ )))
    while p  $\rightarrow$  arr != NULL  $\wedge$  p  $\rightarrow$  arr  $\rightarrow$  dato < p  $\rightarrow$  dato do
        var tmp :  $\alpha$   $\leftarrow$  p  $\rightarrow$  arr  $\rightarrow$  dato
        p  $\rightarrow$  arr  $\rightarrow$  dato  $\leftarrow$  p  $\rightarrow$  dato
        p  $\rightarrow$  dato  $\leftarrow$  tmp
        p  $\leftarrow$  p  $\rightarrow$  arr
    end while
end function

```

Observemos que en el peor caso, se va a iterar hasta que p \rightarrow arr sea NULL. Pero el invariante de representación asegura que el único caso en que esto sucede es que el nodo sea la raíz. 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 $\equiv \{\neg(p = \text{obs NULL})\}$

Post $\equiv \{\text{Se restableció el rep de la estructura de datos}\}$

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( $\alpha$ )))
  while (p->izq != NULL  $\wedge$  p->dato < p->izq->dato) v
    (p->der != NULL  $\wedge$  p->der->dato < p->dato) do
      if p->izq != NULL then
        var tmp :  $\alpha \leftarrow$  p->izq->dato
        p->izq->dato  $\leftarrow$  p->dato
        p->dato  $\leftarrow$  tmp
      else
        if p->der != NULL then
          var tmp :  $\alpha \leftarrow$  p->der->dato
          p->der->dato  $\leftarrow$  p->dato
          p->dato  $\leftarrow$  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: $\wedge(\text{it}(\alpha)) \rightarrow \text{boolean}$

Rep(e) $\equiv \text{true} \iff \neg(e = \text{obs NULL})$

Abs: $\wedge(\text{it}(\alpha)) \rightarrow \text{IteradorUnidireccional}(\alpha)$

($\forall e : \wedge(\text{it}(\alpha))$) Abs(e) =obs i / Siguietes(i) =obs *e • <>

```
*****
*****                               Algoritmos del Iterador                               *****
*****
```

Pre $\equiv \{p \text{ es un puntero en la estructura de } t\}$

Post $\equiv \{\text{itCola es un iterador posicionado en } p\}$

Complejidad: $O(1)$

Descripción: Crea un puntero disfrazado de iterador.

crearIter(in t : colaPrioridad(α), p : puntero(nodo(α))) \rightarrow res : itCola(α)

```
  res  $\leftarrow$  p
end function
```