# 1. Módulo Tried(k, $\sigma$)

# Interfaz

**parámetros formales**
**géneros** k, $\sigma$
**función** $\bullet = \bullet$(**in** $k1$ : k, **in** $k2$ : k) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} (k1 = k2)$}
**Complejidad:** $\Theta(equal(k1, k2))$
**Descripción:** función de comparación de k's
COPIAR(**in** $a$ : k) $\rightarrow$ $res$ : k
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} a$}
**Complejidad:** $\Theta(copy(a))$
**Descripción:** función de copia de k's
COPIAR(**in** $s$ : $\sigma$) $\rightarrow$ $res$ : $\sigma$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} s$}
**Complejidad:** $\Theta(copy(s))$
**Descripción:** función de copia de $\sigma$'s
LONG(**in** $a$ : k) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} long(a)$}
**Complejidad:** $\Theta(1)$
**Descripción:** función que devuelve el tamaño de k's

**se explica con**: DICCIONARIO($k, \sigma$).

**géneros**: `tried`($k, \sigma$).

## Operaciones básicas de tried

VACÍO() $\rightarrow$ $res$ : `tried(k,`$\sigma$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ vacio}
**Complejidad:** $\Theta(1)$
**Descripción:** genera un tried vacío.

DEFINIR(**in/out** $d$: `tried(k,`$\sigma$`)`, **in** $k$ : k, **in** $s$ : $\sigma$ )
**Pre** $\equiv$ {$d =_{\text{obs}} d_0$}
**Post** $\equiv$ {$d =_{\text{obs}} definir(d_0, k, s)$}
**Complejidad:** O(long(k) + copy(k))
**Descripción:** define la clave k con el significado s en el tried d.

DEFINIDO?(**in** $d$: `tried(k,`$\sigma$`)`, **in** $k$ : k) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} def?(d, k)$}
**Complejidad:** O(long(k) + copy(k))
**Descripción:** devuelve true si y sólo k está definido en el tried.

SIGNIFICADO(**in** $d$: `tried(k,`$\sigma$`)`, **in** $k$ : k) $\rightarrow$ $res$ : $\sigma$
**Pre** $\equiv$ {$def?(d, k)$}
**Post** $\equiv$ {$alias(res =_{\text{obs}} obtener(d, k))$}
**Complejidad:** O(long(k) + copy(k))
**Descripción:** devuelve el significado de la clave k en el tried d.
**Aliasing:** res es modificable si y sólo si d es modificable.

BORRAR(**in/out** $d$: `tried(k,`$\sigma$`)`, **in** $k$ : k)
**Pre** $\equiv$ {$d =_{\text{obs}} d_0 \wedge def?(d, k)$}

**Post** $\equiv \{d =_{\text{obs}} borrar(d_0, k)\}$
**Complejidad:** O(Eliminar del conjunto lineal + copy(k) + long(k))
**Descripción:** borra la clave k del tried d.

CLAVES(**in** $d$: `tried(k,`$\sigma$`)`) $\rightarrow$ $res$ : `conj(k)`
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} claves(d)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve las claves del tried.
**Aliasing:** res es modificable si y sólo si d es modificable.

#CLAVES(**in** $d$: `tried(k,`$\sigma$`)`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} \#(claves(d))\}$
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve la cantidad de claves del tried.

# Representación

## Representación de la tried

`tried(k,`$\sigma$`)` **se representa con** `trie`

donde `trie` es `tupla`(*diccionario*: `lista(nodoTrie)`, *claves*: `conj(k)`)
donde `nodoTrie` es `tupla`(*letra*: `k`, *final*: `bool`, *siguientes*: `lista(nodoTrie)`, *significado*: `puntero(`$\sigma$`)`)

Rep : trie $\longrightarrow$ bool
Rep($t$) $\equiv$ true $\iff$ ($\forall i$: nat)((i >0 $\wedge$ i <Longitud(t.diccionario)) $\Rightarrow_{\text{L}}$ long(t.diccionario[i].letra) = 1 $\wedge$ $\neg$LetrasReperidas?(t.diccionario) $\wedge$ (t.final $\Rightarrow$ t.significado $\neq$ NULL)

LetrasReperidas? : lista(nodoTrie) $\longrightarrow$ bool
LetrasReperidas?(l) $\equiv$ **if** EsVacía?(l) **then**
true
**else**
LetrasReperidas?(Fin(l)) $\wedge$ LetrasReperidas?Aux(Fin(l), Primero(l).letra)
**fi**

LetrasReperidas?Aux : lista(nodoTrie) $\times$ k $\longrightarrow$ bool
LetrasReperidas?Aux($l$, $k$) $\equiv$ **if** (EsVacía?(l)) **then**
true
**else**
**if** (Primero(l).letra = k) **then** false **else** LetrasReperidas?Aux(Fin(l), k) **fi**
**fi**

Abs : trie $t$ $\longrightarrow$ tried {Rep($t$)}
Abs($t$) $\equiv$ e
($\forall$ c : k)(def?(e, c) $\Leftrightarrow$ Pertenece?(t.claves, c) $\wedge_{\text{L}}$ (def?(e, c) $\Rightarrow_{\text{L}}$ obtener(e, c) = ObtenerDeEstructura(t.diccionario, c)))

ObtenerDeEstructura : lista(nodoTrie) $\times$ k $\longrightarrow$ $\sigma$

```
ObtenerDeEstructura(l, c) ≡ if Primero(l).letra = prim(k) ∧ vacía?(fin(k)) then
                              *Primero(l).significado
                            else
                              if (Primero(l).letra = prim(k)) then
                                ObtenerDeEstructura(Primero(l).siguientes, fin(k)
                              else
                                ObtenerDeEstructura(Fin(l), k)
                              fi
                            fi
```

# Algoritmos

---

**iVacío**() → *res* : trie

res ← <Vacía(), Vacío()> ▷ Θ(1)

Complejidad: Θ(1)

---

---

**iDefinir**(**in**/**out** $d$ : `trie`, **in** $k$ : `k`, **in** $s$ : $\sigma$)

AgregarRapido(d.claves, k) ▷ O(copy(k))
kAux : k ← copy(k) ▷ O(copy(k))
recorrido : itLista(nodoTrie) ← crearIT(d.diccionario) ▷ O(1)
**while** ¬ vacía?(kAux) **do** ▷ O(8 x long(k)) = O(long(k))
  encontrado : bool ← false ▷ O(1)
  **while** HaySiguiente(recorrido) ∧ ¬ encontrado **do** ▷ O(7 x cantidad de elementos asignados a la lista que itera recorrido) pero esto está acotado por constante así que es O(1)
    **if** Siguiente(recorrido) = prim(kAux) **then** ▷ O(2)
      encontrado ← true ▷ O(1)
    **end if**
    Avanzar(recorrido) ▷ O(1)
  **end while**
  **if** ¬encontrado **then** ▷ O(5)
    nueva : lista(nodoTrie) ← Vacía() ▷ O(1)
    *aAgregar : puntero($\sigma$) ← s ▷ O(1)
    AgregarComoSiguiente(recorrido, <prim(kAux) , vacía?(fin(kAux)), nueva, aAgregar>) ▷ O(copy(<prim(kAux) , vacía?(fin(kAux)), nueva, s>)) = O(1)
    recorrido ← crearIT(Siguiente(recorrido).Siguiente) ▷ O(1)
  **else if** encontrado ∧ vacía?(fin(kAux)) **then** ▷ O(5)
    Siguiente(recorrido).final ← true ▷ O(1)
    *Siguiente(recorrido).significado ← s ▷ O(1)
  **else if** encontrado **then** ▷ O(2)
    recorrido ← crearIT(Siguiente(recorrido).Siguiente) ▷ O(1)
  **end if**
  kAux ← fin(kAux) ▷ O(1)
**end while**

Complejidad: O(long(k) + copy(k))
Justificación: O(long(k) + 1 + 2xcopy(k)) = O(long(k) + copy(k))

---

**iDefinido?**(**in** $d$ : `trie`, **in** $k$ : `k`)→ $res$ : bool

kAux : k ← copy(k) ▷ O(copy(k))
recorrido : itLista(nodoTrie) ← crearIT(d.diccionario) ▷ O(1)
res ← false ▷ O(1)
**while** ¬ vacía?(kAux) ∧ ¬res ∧ HaySiguiente(recorrido) **do** ▷ O(12 x long(k)) = O(long(k))
    encontrado : bool ← false ▷ O(1)
    **while** HaySiguiente(recorrido) ∧ ¬ encontrado **do** ▷ O(7 x cantidad de elementos asignados a la lista que itera recorrido) pero esto está acotado por constante así que es O(1)
        **if** Siguiente(recorrido) = prim(kAux) **then** ▷ O(2)
            encontrado ← true ▷ O(1)
        **end if**
        Avanzar(recorrido) ▷ O(1)
    **end while**
    **if** ¬ encontrado **then** ▷ O(2)
        res ← false ▷ O(1)
    **else if** encontrado ∧ vacía?(fin(kAux)) **then** ▷ O(4)
        res ← Siguiente(recorrido).final ▷ O(1)
    **else if** encontrado **then** ▷ O(2)
        recorrido ← crearIT(Siguiente(recorrido).Siguiente) ▷ O(1)
    **end if**
    kAux ← fin(kAux) ▷ O(1)
**end while**)

Complejidad: O(copy(k) + long(k))
Justificación: O(copy(k) + long(k) + 2) = O(copy(k) + long(k))

---

**iSignificado**(**in** $d$ : `trie`, **in** $k$ : `k`)→ $res$ : $\sigma$

kAux : k ← copy(k) ▷ O(copy(k))
resultado : puntero($\sigma$) ← NULL ▷ O(1)
recorrido : itLista(nodoTrie) ← crearIT(d.diccionario) ▷ O(1)
**while** ¬ vacía?(kAux) **do** ▷ O(6 x long(k)) = O(long(k))
    encontrado : bool ← false ▷ O(1)
    **while** HaySiguiente(recorrido) ∧ ¬ encontrado **do** ▷ O(7 x cantidad de elementos asignados a la lista que itera recorrido) pero esto está acotado por constante así que es O(1)
        **if** Siguiente(recorrido) = prim(kAux) **then** ▷ O(2)
            encontrado ← true ▷ O(1)
        **end if**
        Avanzar(recorrido) ▷ O(1)
    **end while**
    **if** encontrado ∧ vacía?(fin(kAux)) **then** ▷ O(3)
        resultado ← Siguiente(recorrido).significado ▷ O(1)
    **else** ▷ O(2)
        recorrido ← crearIT(Siguiente(recorrido).Siguiente) ▷ O(1)
    **end if**
    kAux ← fin(kAux) ▷ O(1)
**end while**
res ← *resultado ▷ O(1)

Complejidad: O(copy(k) + long(k))
Justificación: O(copy(k) + long(k) + 3) = O(copy(k) + long(k))

**iBorrar**(**in/out** $d$: `trie`, **in** $k$: `k`)

Eliminar(d.claves, k) ▷ $O(\Sigma_{a' \in d.claves} equal(k, a'))$
kAux : k ← copy(k) ▷ O(copy(k))
recorrido : itLista(nodoTrie) ← crearIT(d.diccionario) ▷ O(1)
**while** ¬ vacía?(kAux) **do** ▷ O(4 x long(k)) = O(long(k))
    encontrado : bool ← false ▷ O(1)
    **while** HaySiguiente(recorrido) ∧ ¬ encontrado **do** ▷ O(7 x cantidad de elementos asignados a la lista que itera recorrido) pero esto está acotado por constante así que es O(1)
        **if** Siguiente(recorrido) = prim(kAux) **then** ▷ O(2)
            encontrado ← true ▷ O(1)
        **end if**
        Avanzar(recorrido) ▷ O(1)
    **end while**
    **if** vacía?(fin(kAux)) **then** ▷ O(2)
        Siguiente(recorrido).final ← false ▷ O(1)
    **else** ▷ O(2)
        recorrido ← crearIT(Siguiente(recorrido).Siguiente) ▷ O(1)
    **end if**
    kAux ← fin(kAux) ▷ O(1)
**end while**

Complejidad: O($\Sigma_{a' \in d.claves} equal(k, a')$ + copy(k) + long(k))
Justificación: O($\Sigma_{a' \in d.claves} equal(k, a')$ + copy(k) + long(k) + 1) = O($\Sigma_{a' \in d.claves} equal(k, a')$ + copy(k) + long(k))

---

**Claves**(**in** $d$: `trie`)→ $res$ : conj(k)

res ← d.claves ▷ O(1)

Complejidad: O(1)

---

#**Claves**(**in** $d$: `trie`)→ $res$ : nat

res ← Cardinal(d.claves) ▷ O(1)

Complejidad: O(1)

---

**Aclaraciones:** - Varias de las complejidades incluyen copy(k) haciendo referencia al costo de copiar la clave. Como las claves con las que vamos a usar este modulo son hostname, y por ende secuencias de caracteres, el costo de copia es igual a la longitud de la clave. Entonces estas complejidades nos quedan O(long(k) + long(k)) = O(long(k)).