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

Parametros formales
    generos α
    funcion Copiar(in a: α) → res: α
        Pre ≡ {true}
        Post ≡ {res =obs a}
        Complejidad: O(copy(a))
        Descripcion: funcion de copia de α's.

    funcion • = • (in a1 : α, a2 : α) → res : bool
        Pre ≡ {true}
        Post ≡ {res =obs (a1 = a2)}
        Complejidad: O(equal(a1, a2))
        Descripcion: funcion de igualdad de αs

Se explica con: dicc(string, α)
Generos: DiccString(α)

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

Crear() → res : DiccString(α)
Pre ≡ {true}
Post ≡ {res =obs vacio}
Complejidad: O(1)
Descripcion: Crea un diccionario vacio.

Definir(in/out d : DiccString(α), in k : string, in v : α)
Pre ≡ {d =obs d₀}
Post ≡ {d =obs definir(k, v, d₀)}
Complejidad: O(long(k))
Descripcion: Inserta una clave en el diccionario.

Definido?(in d : DiccString(α), in k : string) → res : bool
Pre ≡ {true}
Post ≡ {res =obs def?(k, d)}
Complejidad: O(long(k))
Descripcion: Dice si una clave esta definido.

Obtener(in d : DiccString(α), in k : string) → res : α
Pre ≡ {def?(k, d)}
Post ≡ {res =obs obtener(k, d)}
Complejidad: O(long(k))
Descripcion: Obtiene el significado de la clave en el diccionario

Claves(in d : DiccString(α)) → res : itConj(string)
Pre ≡ {true}
Post ≡ {alias(esPermutación?(SecuSuby(res), claves(d))) ∧ vacia?(Anteriores(res))}
Complejidad: O(1)
Descripcion: Devuelve un iterador a las claves del diccionario.

Copia(in d: DiccString(α)) → res : DiccString(α)
Pre ≡ {true}
Post ≡ {res =obs d}
Complejidad: O(#Claves(d) * long(e) * Copia(Obtener(d, s))), donde e es la clave mas larga
            y s pertenece a Claves(d)
Descripcion: Crea un diccionario por copia.
```

```
*****************************************************************************
************                    Representación                     ************
*****************************************************************************

DiccString(α) se representa con cabeza(α)
    donde cabeza(α) es tupla(
                        claves: conj(string),
                        significados: trie(α)
                        )

    donde trie(α) es tupla(
                        continuacion : array_estatico[256] de puntero(trie(α)),
                        significado  : puntero(α)
                        )

Rep: cabeza(α) t → bool
Rep(t) ≡ true ⟺
    ((∀s: string)(s ∈ t.claves ⟺ def?'(t.significados, s))) ∧
    No hay ciclos en significados ∧
    ((∀i : nat) (i < 256) ⟹ι t.continuacion[i] =obs NULL) ⟹ t.significado =obs NULL ∧ι
    (∀i : nat) (i < 256 ∧ι ¬(t.continuacion[i] =obs NULL)) ⟹ι Rep(*(t.continuacion[i]))

Abs: ^(cabeza(α)) t → dicc(string, α)                                    {Rep(t)}
(∀t : ^(cabeza(α))) Abs(t) =obs d /
    (∀c : string)
        def?(c, d) =obs def'?(c, t.significados) ∧
        def'?(c, t.significados) ⟹ι obt'(c, t.significados) =obs obtener(c, d)

def?': string × ^(trie(α)) → bool
def?'(s, t) ≡
    if vacia?(s) then ¬(t.significado =obs NULL)
    else
        if t.continuacion[ord(prim(s))] =obs NULL then false
        else def?(fin(s), t.continuacion[ord(prim(s))]) fi
    fi

obt': string s × ^(trie(α)) t → α                                {def?'(s, t)}
obt'(s, t) ≡
    if vacia?(s) then t.significado
    else obt'(fin(s), t.continuacion[ord(prim(s))]) fi

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

iCrear() → res : cabeza(α)
    res ← ⟨claves: Vacio(),
         significados: ⟨continuacion: arreglo_estatico[256] de puntero(trie(α)),
             significado: NULL⟩
        ⟩
end function

iDefinir(in/out d : cabeza(α), in k : string, in v : α)
    var i : nat ← 0
    var t : puntero(trie(α)) ← &d.significados
    var nuevo: bool ← false
    while i < longitud(k) do
        if t→continuacion[ord(k[i])] == NULL then
            t→continuacion[ord(k[i])] ← &Crear()
            nuevo ← true
        end if
        t ← *t→continuacion[ord(k[i])]
        i ← i + 1
    end while
    t→significado ← &v
    if nuevo then
        AgregarRapido(d.claves, k)
    fi
end function
```

Como se puede observar el bucle principal depende exlusivamente de la cantidad de caracteres del vector, por lo que el mejor caso es que el mismo tenga 0 caracteres en donde automaticamente saldra y definira la palabra vacia y su costo sera O(1), pero tanto el caso promedio como el peor sera en donde el largo del vector es mayor a 0 entonces dado que todas las operaciones dentro del bucle tienen costo O(1) tendra un costo equivalente al tamaño del vector (O(long(k))) ya que tendra que recorrerlo para definir letra por letra dentro de la estructura hasta llegar a su fin y definir el significado que tiene costo O(1). A esto se le suma la complejidad de AgregarRapido de la clave al conjunto de claves (si esta no estaba definida), al ser la clave un string esto cuesta el copiar del vector de char, que es la longitud del vector por el costo de copiar cada char que es 1, quedando entonces la longitud del string k.

```
iDefinido?(in d : cabeza(α), in k : string) → res : bool
    var i : nat ← 0
    var t : puntero(trie(α)) ← &d.significados
    bool listo ← False
    while i < longitud(k) ∧ t→continuacion[ord(k[i])] != NULL do
        t ← t→continuacion[ord(k[i])]
        i ← i + 1
    end while
    res ← t→significado != NULL ∧ i == longitud(k)
end function
```

En el peor caso(si el string esta definido dentro de la estructura y el mismo tiene largo mayor a 0) dado que las operaciones dentro del bucle tienen costo O(1) y se repetiran long(k) veces, para ir recorriendo la estructura, este tendra un costo O(long(k)) pero en el caso de que no este definido el costo sera O(long(k) - n) con n cantidad de caracteres que no pertenecen al prefijo perteneciente a la estructura por lo que el peor caso sera O(long(k)).

```
iObtener(in d : cabeza(α), in k : string) → res : α
    var i : nat ← 0
    var t : puntero(trie(α)) ← &d.significados
    while i < longitud(k) do
        t ← t→continuacion[ord(k[i])]
        i ← i + 1
    end while
    res ← *t→significado
end function
```

En el peor caso dado que las operaciones dentro del bucle tienen costo O(1) y se repetiran long(k) veces, para ir recorriendo la estructura, este tendra un costo O(long(k)).

```
iClaves(in d: cabeza(α)) → res : itConj(string)
    res ← CrearIt(d.claves)
end function
```

```
iCopia(in d: cabeza(α)) → res : cabeza(α)
    var i : nat ← 0
    var c : cabeza(α) ← Crear()
    var it : itConj(string) ← CrearIt(d.claves)
    while HaySiguiente(it)
        Definir(c, Siguiente(it), Copiar(Obtener(d, Siguiente(it))))
        Avanzar(it)
    end while
    res ← c
end function
```

Al copiar se itera por todas las claves y se definen los significados creando una copia de cada uno. Por lo tanto se multiplica el cardinal del conjunto de claves por el obtener de cada claves por el copiar de cada significado. Por lo tanto para el peor caso tomamos el peor obtener que seria el de la clave mas larga y la copia de todos los significados.