1. Modulo DiccRapido(clave, significado)

Interfaz

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO)

generos: `diccRapido(nat, significado)`

Operaciones basicas de conjunto

VACIO(**in** *maxClaves*: `nat`) → *res* : `diccRapido(nat, significado)`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ vacio()}
Complejidad: O(N) donde N es *maxClaves*
Descripción: Crea un nuevo diccionario vacio.

DEFINIR(**in** *k*: `clave` **in** *s*: `significado`, **in/out** *d*: `diccRapido(nat, significado)`)
Pre ≡ {d =$_{\text{obs}}$ d_0}
Post ≡ {*d* =$_{\text{obs}}$ definir(*k*, *s*, d_0)}
Complejidad: O(N) en peor caso. O(1) en el caso promedio si se asegura distribucion uniforme de las claves.
Descripción: Define el elemento *k*, con significado *s*, en el diccionario *d*

DEF?(**in** *c*: `clave`, **in** *d*: `diccRapido(nat, significado)`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ def?(*c*, *d*)}
Complejidad: O(N) en peor caso. O(1) en el caso promedio si se asegura distribucion uniforme de las claves.
Descripción: Devuelve *true* si la clave *k* esta definida

OBTENER(**in** *c*: `clave`, **in** *d*: `diccRapido(nat, significado)`) → *res* : `significado`
Pre ≡ {def?(*c*, *d*)}
Post ≡ {alias(*res* =$_{\text{obs}}$ obtener(*c*, *d*))}
Complejidad: O(N) en peor caso. O(1) en el caso promedio si se asegura distribucion uniforme de las claves.
Descripción: Devuelve el significado de la clave *c*
Aliasing: *res* es una referencia al significado de *c* en el diccionario *d*. Si se modifica, se modificara el significado dentro del diccionario. Si se borra una clave, o se define alguna clave, la referencia queda invalidada.

BORRAR(**in** *c*: `clave`, **in/out** *d*: `diccRapido(nat, significado)`)
Pre ≡ {def?(*c*, *d*)}
Post ≡ {*res* =$_{\text{obs}}$ borrar(*c*, *d*)}
Complejidad: O(N) en peor caso. O(1) en el caso promedio si se asegura distribucion uniforme de las claves.
Descripción: Devuelve el significado de la clave *c*
Aliasing: *res* es una referencia al significado de *c* en el diccionario *d*. Si se modifica, se modificara el significado dentro del diccionario. Si se borra una clave, o se define alguna clave, la referencia queda invalidada.

CLAVES(**in** *d*: `diccRapido(nat, significado)`) → *res* : `conj(clave)`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ claves(*d*)}
Complejidad: Θ(1)
Descripción: Devuelve el conjunto de claves del diccionario *d*
Aliasing: *res* es una referencia constante a un conj(clave).

Representación

Representacion del DiccRapido

`diccRapido(nat, significado)` **se representa con** `estr`

donde `estr` es `tupla`(*defs*: `arreglo(conj(tuplaSignificado))` , *claves*: `conj(nat)`)
donde `tuplaSignificado` es `tupla`(*key*: `nat` , *def*: `significado`)

Invariante de representacion en castellano:

1. Todas las claves estan definidas en el arreglo y todas las cosas definidas estan en claves.

2. Para toda clave c, la funcion de Hash convierte a c en una posicion del arreglo en donde existe un elemento cuya clave es c.

Rep : estr $\longrightarrow$ bool
Rep(d) $\equiv$ true $\Longleftrightarrow$

1. (($\forall$ c : nat) (Pertenece?(c, d.claves) $\Longleftrightarrow$ ($\exists!$ i : nat) Definido?(i, d.defs) $\wedge_{\text{L}}$ ($\exists$ s : significado) Pertenece?($\langle$ c, s $\rangle$, d.defs[i]))) $\wedge_{\text{L}}$

2. (($\forall$ c : nat) Pertenece?(c, d.claves) $\Rightarrow$ (Definido?(Hash(c, d), d.defs) $\wedge_{\text{L}}$ (($\exists$ s : significado) Pertenece?($\langle$ c, s $\rangle$, d.defs[Hash(c, d)]))))

Hash(c, d) $\equiv$ (c % #(d.claves))

Abs : estr e $\longrightarrow$ Diccionario(clave, significado) {Rep(e)}
Abs(e) $\equiv$ d : Diccionario(clave, significado) /
($\forall$ c : clave) Pertenece?(c, e.claves) $\Longleftrightarrow$ def?(c, d) $\wedge_{\text{L}}$
($\forall$ c : clave) def?(c, d) $\Rightarrow_{\text{L}}$
(($\forall$ s : significado) s $=_{\text{obs}}$ obtener(c, d) $\Longleftrightarrow$ Pertenece?($\langle$ c, s $\rangle$, d.defs[Hash(c, d)]))

Algoritmos

Algoritmos de Agentes

Lista de algoritmos


```
iVacio(in maxClaves : nat) → res: estr
begin
    var
    i : nat
    res.claves ← Vacio()                                          //O(1)
    res.defs ← CrearArreglo(maxClaves)                            //O(N)
    i ← 0                                                         //O(1)
    while i < maxClaves do                                        //O(1)
        res.defs[i] ← Vacio()                                     //O(1)
        i ← i + 1                                                 //O(1)
    end
                                                                  //While: O(N)
end
```

Complejidad: O(N) con N = maxClaves

Algoritmo 1: Vacio

```
iDefinir(in k: clave, in s: significado, in/out d: estr)
begin
    var
    i : nat
    posArreglo : nat
    itConjunto : itConj(tuplaSignificado)
    encontrado : bool
    posArreglo ← Hash(k, d)                                          //O(1)
    itConjunto ← CrearIt(d.defs[posArreglo])                         //O(1)
    encontrado ← false                                               //O(1)
    while HaySiguiente(itConjunto) do                                //O(1)
        if Siguiente(itConjunto).key == k then                       //O(1)
            encontrado ← true                                        //O(1)
            Siguiente(itConjunto).def ← s                            //O(1)
        else
            Avanzar(itConjunto)                                      //O(1)
        end
    end
                                                                     //While: O(N)
    if encontrado == false then                                      //O(1)
        Agregar(d.claves, k)                                         //O(1)
        Agregar(d.defs[posArreglo], ⟨ k, s ⟩)                        //O(1)
    end
end
```

Complejidad: O(N) | O(1)

Comentarios: El peor caso ocurre cuando la funcion de Hash envia a todas las claves a la misma posicion del arreglo. Si se asegura buena distribucion de las claves, se obtiene un buen factor de carga, y la complejidad del *while* se vuelve O(1) en el caso promedio (Pocos elementos en cada lugar del arreglo).

Algoritmo 2: Definir

```
iDef?(in k: clave, in d: estr) → res: bool
begin
    res ← Pertenece?(k, d.claves)                                    //O(N)
end
```

Complejidad: O(N)

Algoritmo 3: Def?

```
iObtener(in k: clave, in d: estr) → res: significado
begin
    var
    i : nat
    posArreglo : nat
    itConjunto : itConj(tuplaSignificado)
    posArreglo ← Hash(k, d)                                          //O(1)
    itConjunto ← CrearIt(d.defs[posArreglo])                         //O(1)
    while HaySiguiente(itConjunto) do                                //O(1)
        if Siguiente(itConjunto).key == k then                       //O(1)
            res ← Siguiente(itConjunto).def                          //O(1)
        else
            Avanzar(itConjunto)                                      //O(1)
        end
    end
                                                                     //While: O(N)
end
```

Complejidad: O(N) | O(1)

Comentarios: El peor caso ocurre cuando la funcion de Hash envia a todas las claves a la misma posicion del arreglo. Si se asegura buena distribucion de las claves, se obtiene un buen factor de carga, y la complejidad del *while* se vuelve O(1) en el caso promedio (Pocos elementos en cada lugar del arreglo).

Algoritmo 4: Obtener

```
iClaves(in d: estr) → res: conj(clave)
begin
    res ← d.claves                                                   //O(1)
end
```

Complejidad: O(1)

Algoritmo 5: Claves

```
Hash(in c: clave, in d: estr) → res: nat
begin
    res ← (c % Cardinal(d.claves) )                                  //O(1)
end
```

Complejidad: O(1)

Algoritmo 6: Hash

```
iBorrar(in k: clave, in/out d: estr)
begin
    var
    i : nat
    posArreglo : nat
    itConjunto : itConj(tuplaSignificado)
    posArreglo ← Hash(k, d)                                          //O(1)
    itConjunto ← CrearIt(d.defs[posArreglo])                         //O(1)
    while HaySiguiente(itConjunto) do                                //O(1)
        if Siguiente(itConjunto).key == k then                       //O(1)
            EliminarSiguiente(itConjunto)                            //O(1)
        else
            Avanzar(itConjunto)                                      //O(1)
        end
    end
                                                                     //While: O(N)
    Eliminar(d.claves, k)                                            //O(N)
end
Complejidad: O(N)
```

Algoritmo 7: Borrar