1. Modulo DiccRapido(clave, significado)

Interfaz

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO)

generos: `diccRapido(clave, significado)`

Operaciones basicas de conjunto

clave debe ser de tipo nat, sino la funcion de hash no funciona.

VACIO() $\rightarrow$ *res* : `diccRapido(clave, significado)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacio()}
Complejidad: O(N) donde N es la cantidad de claves en c
Descripción: Crea un nuevo diccionario vacio.

DEFINIR(**in** k: `clave` **in** s: `significado`, **in/out** d: `diccRapido(clave, significado)`)
Pre $\equiv$ {d $=_{\text{obs}}$ d_0}
Post $\equiv$ {$d =_{\text{obs}}$ definir(k, s, d_0)}
Complejidad: $\Theta(1)$
Descripción: Define el elemento k, con significado s, en el diccionario d

DEF?(**in** c: `clave`, **in** d: `diccRapido(clave, significado)`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ def?(c, d)}
Complejidad: O(N), con N siendo la cantidad de claves
Descripción: Devuelve *true* si la clave k esta definida

OBTENER(**in** c: `clave`, **in** d: `diccRapido(clave, significado)`) $\rightarrow$ *res* : `significado`
Pre $\equiv$ {def?(c, d)}
Post $\equiv$ {alias(*res* $=_{\text{obs}}$ obtener(c, d))}
Complejidad: $\Theta(1)$
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(clave, significado)`) $\rightarrow$ *res* : `conj(clave)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ dameUno(c)}
Complejidad: $\Theta(1)$
Descripción: Devuelve el conjunto de claves del diccionario d
Aliasing: *res* es una referencia constante a un conj(clave).

Representación

Representacion del DiccRapido

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

donde `estr` es `tupla`(*defs*: `arreglo(tuplaSignificado)` , *claves*: `conj(clave)`)
donde `tuplaSignificado` es `tupla`(*key*: `clave` , *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. La funcion Hash manda a cada clave al lugar donde esta definida.

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

1. (($\forall$ c : clave) En(c, d.claves) $\Rightarrow$ (($\exists!$ i : nat) Definido?(i, d.defs) $\wedge_{\text{L}}$ d.defs[i].key = c)) $\wedge$
   (($\forall$ i : nat) Definido?(i, d.defs) $\Rightarrow_{\text{L}}$ En(d.defs[i].key, d.claves)) $\wedge_{\text{L}}$
2. (($\forall$ c : clave) En(c, d.claves) $\Rightarrow$ (Definido?(Hash(c, c, d), d.defs) $\wedge_{\text{L}}$ d.defs[Hash(c, c, d)].key = c)
)

Hash(k, c, d) $\equiv$ **if** (Definido?(fHash) $\wedge_{\text{L}}$ d.defs[fHash].key == c) $\vee$ ($\neg$Definido?(fHash)) **then**
fHash
else
Hash($k + 1$, c, d)
fi
//Donde fHash es (k % Cantidad(d.claves))
//La clave k es el valor sobre el cual aplico el Hash, y la clave c es la clave que busco ubicar.

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

Algoritmos

Algoritmos de Agentes

Lista de algoritmos

iVacio() $\rightarrow$ res: estr
begin
res.claves $\leftarrow$ Vacio() //**O(1)**
res.defs $\leftarrow$ CrearArreglo(1) //**O(1)**
end
Complejidad: O(1)

Algoritmo 1: Vacio

```
iDefinir(in k: clave, in s: significado, in/out d: estr)
begin
    var
    i : nat
    arregloAux : Arreglo
    i ← 0                                                                          //O(1)
    if Def?(k, d) then                                                             //O(1)
        //Entonces es redefinir
        d.defs[Hash(k, k, d)].def ← s                                              //O(1)
        d.defs[Hash(k, k, d)].key ← k                                              //O(1)
    else
        //Entonces es definir algo nuevo, asi que debo ampliar el arreglo
        Agregar(s, d.claves)                                                       //O(N)
        arregloAux ← CrearArreglo(Tam(d.defs) +1 )                                 //O(N)
        while i < Tam(d.defs) do                                                   //O(1)
            //Re-Hasheo con la nueva cantidad de keys
            arregloAux[Hash(d.defs[i].key, d.defs[i].key, d)].key ← d.defs[i].key  //O(1)
            arregloAux[Hash(d.defs[i].key, d.defs[i].key, d)].def ← d.defs[i].def  //O(1)
            i ← i + 1                                                              //O(1)
        end
                                                                                   //While: O(N)
        //Y defino a k donde corresponde
        arregloAux[Hash(k, k, d)].def ← s                                          //O(1)
        arregloAux[Hash(k, k, d)].key ← k                                          //O(1)
        d.defs ← arregloAux                                                        //O(N)
    end
end
Complejidad: O(N)
```

Algoritmo 2: Definir

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

Algoritmo 3: Def?

```
iObtener(in k: clave, in d: estr) → res: significado
begin
    res ← &(d.defs[Hash(k, k, d)].significado)                                     //O(1)
end
Complejidad: O(1)
```

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
| var
| k : nat
| desvio : nat
| desvio ← 0 //**O(1)**
| k ← c % $Cardinal(d.claves)$ //**O(1)**
| **while** *Definido?(k, d.defs)* $\wedge_{\text{L}}$ $\neg$*(d.defs[k].key* = *c)* **do** //**O(1)**
| | desvio ← desvio + 1 //**O(1)**
| | k ← $(c + desvio)$ % $Cardinal(d.claves)$ //**O(1)**
| **end**
| //**While: O(1) en caso promedio. O(N) peor caso.** res ← k //**O(1)**
end

Complejidad: O(1) en caso Promedio con buena distribucion. O(N) peor caso.

Comentarios: La idea es que intento poner a c en el resto de c en la division por la cantidad de claves. Si fallo, intento ponerlo en el siguiente lugar, y asi hasta poder colocarlo.

Algoritmo 6: nombre