Algoritmos y Estructuras de Datos

Apunte **TADs básicos y sus implementaciones**

Segundo Cuatrimestre 2025

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Este documento contiene la especificación de los TADs básicos y, para cada uno, los módulos que los implenentan. Todos se pueden se pueden usar en la resolución de ejercicios de la segunda mitad de la materia (prácticas 6 y 7 y en el parcial) sin tener que definirlos. Nos van a servir para implementar estructuras y módulos más complejos.

Índice

1. Secuencia

1.1. Especificación

TAD Secuencia$\langle T \rangle$ {

obs elems : seq$\langle T \rangle$

proc secuenciaVacía() : Secuencia$\langle T \rangle$ {
asegura { $res.elems = \langle\rangle$ }
}

proc igualdad(in S_1 : Secuencia$\langle T \rangle$, in S_2 : Secuencia$\langle T \rangle$) : bool {
asegura {
$res = \text{true} \leftrightarrow |S_1.elems| = |S_2.elems| \wedge_L$
$(\forall i : \mathbb{Z})\ (0 \leq i < |S_1.elems| \rightarrow_L S_1.elems[i] = S_2.elems[i])$
}
}

proc agregarAdelante(inout s : Secuencia$\langle T \rangle$, in $e : T$) {
requiere { $s = S_0$ }
asegura { $s.elems = \langle e \rangle +\!\!+ S_0.elems$ }
}

proc agregarAtrás(inout s : Secuencia$\langle T \rangle$, in $e : T$) {
requiere { $s = S_0$ }
asegura { $s.elems = S_0.elems +\!\!+ \langle e \rangle$ }
}

proc vacía(in s : Secuencia$\langle T \rangle$) : bool {
asegura { $res = true \leftrightarrow s.elems = \langle\rangle$ }
}

proc fin(inout s : Secuencia$\langle T \rangle$) {
requiere { $s = S_0 \wedge |s.elems| > 0$ }
asegura { $s.elems = tail(S_0.elems)$ }
}

proc comienzo(inout s : Secuencia$\langle T \rangle$) {
requiere { $s = S_0 \wedge |s.elems| > 0$ }
asegura { $s.elems = head(S_0.elems)$ }
}

proc primero(in s : Secuencia$\langle T \rangle$) : T {
requiere { $|s.elems| > 0$ }
asegura { $res = s.elems[0]$ }
}

proc último(in s : Secuencia$\langle T \rangle$) : T {
requiere { $|s.elems| > 0$ }
asegura { $res = s.elems[|s.elems| - 1]$ }
}

proc longitud(in s : Secuencia$\langle T \rangle$) : $\mathbb{Z}$ {
asegura { $res = |s.elems|$ }
}

```
    proc obtener(in s : Secuencia⟨T⟩, in i : ℤ) :T {
        requiere { 0 ≤ i < |s.elems| }
        asegura { res = s.elems[i] }
    }
    proc eliminar(inout s : Secuencia⟨T⟩, in i : ℤ) {
        requiere { s = S_0 ∧ 0 ≤ i < |s.elems| }
        asegura { s.elems = subseq(S_0.elems ++ 0, i − 1, subseq(S_0.elems, i + 1, |S_0.elems|)) }
    }
    proc copiar(in s : Secuencia⟨T⟩) : Secuencia⟨T⟩ {
        asegura { res.elems = s.elems }
    }
    proc modificarPosición(inout s : Secuencia⟨T⟩, in i : ℤ, in e : T) {
        requiere { s = S_0 ∧ 0 ≤ i < |s.elems| }
        asegura { s.elems = subseq(S_0.elems ++ 0, i − 1, ⟨e⟩, subseq(S_0.elems, i + 1, |S_0.elems|)) }
    }
    proc concatenar(inout s : Secuencia⟨T⟩, in s' : Secuencia⟨T⟩) {
        requiere { s = S_0 }
        asegura { s.elems = S_0.elems ++ s'.elems }
    }
}
```

1.2. Implementaciones

1.2.1. Lista Enlazada

El módulo Lista Enlazada[1] implementa el TAD Secuencia. Su ventaja es que el acceso al primer y último elemento para lectura, inserción, modificación y borrado es eficiente (costo constante). Por el contrario, acceder en forma directa a un elemento arbitrario ("acceso directo" o "acceso aleatorio") tiene un costo lineal. En memoria se implementa con una *lista doblemente enlazada*, con referencias al primer y último elementos. Su estructura se apoya en el tipo NodoLista, que contiene el dato a guardar y punteros al próximo nodo y al anterior.

La complejidad de cada una de sus operaciones, de acuerdo a lo definido en el TAD Secuencia, es la siguiente:

proc	complejidad
`secuenciaVacía`	$O(1)$
`longitud`	$O(1)$
`vacía`	$O(1)$
`agregarAdelante`	$O(1)$
`agregarAtrás`	$O(1)$
`fin`	$O(1)$
`comienzo`	$O(1)$
`primero`	$O(1)$
`último`	$O(1)$
`obtener`	$O(n)$
`eliminar`	$O(n)$
`copiar`	$O(n)$
`modificarPosicion`	$O(n)$
`concatenar`	$O(m)$

donde n es la cantidad de elementos de la lista, m es la cantidad de elementos de la segunda lista interviniente (si la hubiera). Si el costo de copiar un elemento o de comparar dos elementos no fuera constante, se debe tener en cuenta en la complejidad de las operaciones.

[1]Los dos módulos que implementan Secuencia son la excepción a la regla ya que llevan el nombre de la estructura en memoria directamente: usamos `ListaEnlazada` y `Vector` en vez de `SecuenciaLista` y `SecuenciaVector`

La estructura de este módulo es:

```
NodoLista<T> es struct<
    val: T,
    siguiente: NodoLista,
    anterior: NodoLista
>

Módulo ListaEnlazada <T> implementa Secuencia⟨T⟩ <
  var primero: NodoLista<T> /* puntero al primer elemento */
  var último: NodoLista<T> /* puntero al último elemento */
  var longitud: int /* cantidad total de elementos */

  // . . .
>
```

El puntero al primer elemento, junto con el campo `siguiente` de la estructura `NodoLista` nos permite iterar en orden creciente e insertar adelante en O(1). El puntero al último elemento, por otra parte, nos permite iterar de atrás para adelante e insertar al final. Finalmente el campo `longitud` nos permite conocer el tamaño en O(1).

1.2.2. Vector

El módulo Vector provee una secuencia que permite obtener el i-ésimo elemento de forma eficiente. La inserción de elementos es eficiente cuando se realiza al final de la misma, si se utiliza un análisis amortizado (i.e., n inserciones consecutivas cuestan $O(n)$), aunque puede tener un costo lineal en peor caso. La inserción en otras posiciones no es tan eficiente, ya que requiere varias copias de elementos. El borrado de los últimos elementos es eficiente, no así el borrado de los elementos intermedio.

Para describir la complejidad de las operaciones, vamos a utilizar $f(n)$ para denotar las operaciones que tienen un costo $O(1)$ *en términos asintóticos*, es decir, cuando se ejecutan muchas veces. La definición detallada de esta $f(n)$ está en el apéndice de este documento.

Las complejidades de sus operaciones son por lo tanto:

proc	complejidad
`secuenciaVacía`	$O(1)$
`longitud`	$O(1)$
`vacía`	$O(1)$
`agregarAdelante`	$O(f(n))$
`agregarAtrás`	$O(f(n))$
`fin`	$O(n)$
`comienzo`	$O(n)$
`primero`	$O(1)$
`último`	$O(1)$
`obtener`	$O(1)$
`eliminar`	$O(n)$
`modificarPosición`	$O(1)$
`concatenar`	$O(m)$

donde n es la cantidad de elementos del vector, y m es la cantidad de elementos del segundo vector interviniente (si lo hubiera). Si el costo de copiar un elemento o de comparar dos elementos no fuera constante, se debe tener en cuenta en la complejidad de las operaciones.

La implementación por la que optamos es un `Array`, cuyo tamaño irá incrementando a medida que se agregan nuevos elementos. Así, se podrá acceder en forma directa al i-ésimo elemento en $O(1)$.

Para agregar elementos tome $O(1)$ en forma amortizada (i.e., $O(f(n))$ operaciones) podemos duplicar el tamaño del arreglo cuando este se llena, hacer la copia correspondiente de los elementos anteriores y agregar el nuevo al final.

Módulo Vector<T> **implementa** Secuencia$\langle T \rangle$ <

```
  var array: Array<T> /* Elementos de la secuencia */
  var longitud: int /* cantidad total de elementos */

  // . . .
```

>

2. Pila

2.1. Especificación

TAD Pila$\langle T \rangle$ {

obs s : seq$\langle T \rangle$

proc pilaVacía() : Pila$\langle T \rangle$ {
asegura { $res.elems = \langle\rangle$ }
}

proc igualdad(in P_1 : Pila$\langle T \rangle$, in P_2 : Pila$\langle T \rangle$) : bool {
asegura {
$res = \text{true} \leftrightarrow |P_1.elems| = |P_2.elems| \wedge_L$
$(\forall i : \mathbb{Z})\ (0 \leq i < |P_1.elems| \rightarrow_L P_1.elems[i] = P_2.elems[i])$
}
}

proc vacía(in p : Pila$\langle T \rangle$) : bool {
asegura { $res = true \leftrightarrow p.elems = \langle\rangle$ }
}

proc apilar(inout p : Pila$\langle T \rangle$, in $e : T$) {
requiere { $p = P_0$ }
asegura { $p.elems = P_0.elems +\!+ \langle e \rangle$ }
}

proc desapilar(inout p : Pila$\langle T \rangle$) : T {
requiere { $p = P_0 \wedge p.elems \neq \langle\rangle$ }
asegura {
$p.elems = subseq(P_0.elems, 0, |P_0.elems| - 1) \wedge$
$res = P_0.elems[|P_0.elems| - 1]$
}
}

proc tope(in p : Pila$\langle T \rangle$) : T {
requiere { $p.elems \neq \langle\rangle$ }
asegura { $res = p.elems[|p.elems| - 1]$ }
}

}

2.2. Implementación

2.2.1. PilaSobreLista

El módulo PilaSobreLista implementa el TAD Pila. Provee una pila en la que sólo se puede acceder al tope de la misma. Se implementa con una `ListaEnlazada` lo que nos permite realizar todas las operaciones en $O(1)$.

proc	complejidad
pilaVacía	$O(1)$
vacía	$O(1)$
apilar	$O(1)$
desapilar	$O(1)$
tope	$O(1)$

Si el costo de copiar un elemento o de comparar dos elementos no fuera constante, se debe tener en cuenta en la complejidad de las operaciones.

Como mencionamos, su representación es simplemente una ListaEnlazada:

```
Módulo PilaSobreLista<T> implementa Pila⟨T⟩ <
  var pila: ListaEnlazada<T>

  // . . .
>
```

3. Cola

3.1. Especificación

TAD Cola$\langle T \rangle$ {

obs elems : seq$\langle T \rangle$

proc colaVacía() : Cola$\langle T \rangle$ {
asegura { $res.elems = \langle\rangle$ }
}

proc igualdad(in C_1 : Cola$\langle T \rangle$, in C_2 : Cola$\langle T \rangle$) : bool {
asegura {
$res = \text{true} \leftrightarrow |C_1.elems| = |C_2.elems| \land_L$
$(\forall i : \mathbb{Z})\ (0 \leq i < |C_1.elems| \rightarrow_L C_1.elems[i] = C_2.elems[i])$
}
}

proc vacía(in c : Cola$\langle T \rangle$) : bool {
asegura { $res = true \leftrightarrow c.elems = \langle\rangle$ }
}

proc encolar(inout c : Cola$\langle T \rangle$, in $e : T$) {
requiere { $c = C_0$ }
asegura { $c.elems = C_0.elems +\!\!+ \langle e \rangle$ }
}

proc desencolar(inout c : Cola$\langle T \rangle$) : T {
requiere { $c = C_0 \land c.elems \neq \langle\rangle$ }
asegura { $c.elems = subseq(C_0.elems, 1, |C_0.elems|) \land res = C_0.elems[0]$ }
}

proc proximo(in c : Cola$\langle T \rangle$) : T {
requiere { $c.elems \neq \langle\rangle$ }
asegura { $res = c.elems[0]$ }
}

}

3.2. Implementación

3.2.1. ColaSobreLista

En forma análoga a la Pila, el módulo ColaSobreLista implementa el TAD Cola utilizando una `ListaEnlazada` como representación. No incluye iteradores y todas sus operaciones son $O(1)$.

proc	complejidad
colaVacía	$O(1)$
vacía	$O(1)$
encolar	$O(1)$
desencolar	$O(1)$
próximo	$O(1)$

Si el costo de copiar un elemento o de comparar dos elementos no fuera constante, se debe tener en cuenta en la complejidad de las operaciones.

Como mencionamos, su representación es simplemente una ListaEnlazada:

```
Módulo ColaSobreLista <T> implementa Cola⟨T⟩ <
  var cola: ListaEnlazada<T>

  // . . .
>
```

4. Cola de Prioridad

4.1. Especificación

Especificación de una cola de prioridad donde la prioridad es el máximo, según algún criterio de orden. La especificación de una cola de prioridad por mínimo es análoga.

```
TAD ColaPrioridadMax⟨T⟩ {
  obs elems : dicc⟨T, ℤ⟩
  proc colaPrioridadVacía() : ColaPrioridadMax⟨T⟩ {
    asegura { res.elems = ⟨⟩ }
  }
  proc igualdad(in C1 : ColaPrioridadMax⟨T⟩, in C2 : ColaPrioridadMax⟨T⟩) : bool {
    asegura {
      res = true ↔ |C1.elems| = |C2.elems| ∧L
          (∀k : T) (t ∈ C1.elems ↔ t ∈ C2.elems) ∧
          (∀k : T) (t ∈ C1.elems →L C1.elems[t] = C2.elems[t])
    }
  }
  proc vacía(in c : ColaPrioridadMax⟨T⟩) : bool {
    asegura { res = true ↔ c.elems = ⟨⟩ }
  }
  proc encolar(inout c : ColaPrioridadMax⟨T⟩, e : T, in pri : ℤ) {
    requiere { c = C0 ∧ e ∉ c.elems }
    asegura { c.elems = setKey(C0.elems, e, pri) }
  }
```

```
    proc desencolarMax(inout c : ColaPrioridadMax⟨T⟩) : T {
        requiere { c = C_0 ∧ c.elems ≠ ⟨⟩ }
        asegura { tienePrioridadMaxMax(C_0.elems, res) ∧ c.elems = delKey(C_0.elems, res) }
    }

    pred tienePrioridadMax(d : dicc⟨T, ℤ⟩, e : T) {
        e ∈ d ∧_L (∀e' : T) (e' ∈ d →_L d[e] ≥ d[e']))
    }
}
```

4.2. Implementación

4.2.1. ColaDePrioridadLog

El módulo `ColaDePrioridadLog` implementa el TAD `ColaPrioridad` utilizando un *Heap*. Provee todas las operaciones de una cola de prioridad. Es posible construir un ColaDePrioridadLog a partir de una secuencia en $O(n)$ utilizando el algoritmo `heapify`

Las complejidades de las operaciones son las siguientes:

ColaDePrioridadLog

proc	complejidad
`colaDePriodidadVacía`	$O(1)$
`colaDePrioridadDesdeSecuencia`	$O(n)$
`encolar`	$O(\log n)$
`consultarMax`	$O(1)$
`desencolarMax`	$O(\log n)$
`tamaño`	$O(1)$

donde n es la cantidad de elementos.

La implementación, como dijimos, es un *Heap*.

```
Módulo ColaDePrioridadHeap<T> implementa ColaDePrioridad⟨T⟩ <
  var elementos: Heap<T>
  var tamaño: int

  // . . .
>
```

5. Conjunto

5.1. Especificación

```
TAD Conjunto⟨T⟩ {
    obs elems : conj⟨T⟩
    proc conjVacío() : Conjunto⟨T⟩ {
        asegura { res.elems = ⟨⟩ }
    }

    proc igualdad(in C_1 : Conjunto⟨T⟩, in C_2 : Conjunto⟨T⟩) : bool {
        asegura {
            res = true ↔ |C_1.elems| = |C_2.elems| ∧_L
                (∀e : T) (e ∈ C_1.elems ↔ e ∈ C_2.elems[i])
        }
```

```
    }
    proc pertenece(in c : Conjunto⟨T⟩, in e : T) : bool {
        asegura { res = true ↔ e ∈ c.elems }
    }
    proc agregar(inout c : Conjunto⟨T⟩, in e : T) {
        requiere { c = C_0 }
        asegura { c.elems = C_0.elems ∪ ⟨e⟩ }
    }
    proc sacar(inout c : Conjunto⟨T⟩, in e : T) {
        requiere { c = C_0 }
        asegura { c.elems = C_0.elems − ⟨e⟩ }
    }
    proc unir(inout c : Conjunto⟨T⟩, in c' : Conjunto⟨T⟩) {
        requiere { c = C_0 }
        asegura { c.elems = C_0.elems ∪ c'.elems }
    }
    proc restar(inout c : Conjunto⟨T⟩, in c' : Conjunto⟨T⟩) {
        requiere { c = C_0 }
        asegura { c.elems = C_0.elems − c'.elems }
    }
    proc intersecar(inout c : Conjunto⟨T⟩, in c' : Conjunto⟨T⟩) {
        requiere { c = C_0 }
        asegura { c.elems = C_0.elems ∩ c'.elems }
    }
    proc agregarRápido(inout c : Conjunto⟨T⟩, in e : T) {
        requiere { c = C_0 ∧ e ∉ c.elems }
        asegura { c.elems = C_0.elems ∪ ⟨e⟩ }
    }
    proc tamaño(in c : Conjunto⟨T⟩) : ℤ {
        asegura { res = |c.elems| }
    }
}
```

5.2. Implementaciones

5.2.1. ConjuntoLineal

El módulo `ConjuntoLineal` (o `ConjuntoLista`) implementa el TAD Conjunto en el que las operaciones se puede insertar, eliminar, y testear pertenencia en tiempo lineal (de comparaciones y/o copias) y la operación de agregarRápido en tiempo constante.

Las complejidades de las operaciones son las siguientes:

ConjuntoLineal

proc	complejidad
`conjVacío`	$O(1)$
`tamaño`	$O(1)$
`pertenece`	$O(n)$
`agregar`	$O(n)$
`agregarRápido`	$O(1)$
`sacar`	$O(n)$
`unir`	$O(n \times m)$
`restar`	$O(n \times m)$
`intersecar`	$O(n \times m)$

donde n es la cantidad de elementos del conjunto, m es la cantidad de elementos del segundo conjunto interviniente (si lo hubiera). Si el costo de copiar un elemento o de comparar dos elementos no fuera constante, se debe tener en cuenta en la complejidad de las operaciones.

La implementación es una lista enlazada.

```
Módulo ConjuntoLineal<T> implementa Conjunto⟨T⟩ <
  var elementos: ListaEnlazada<T>
  var tamaño: int

  // . . .
>
```

5.2.2. ConjuntoLog

El módulo `ConjuntoLog` (o `ConjuntoAVL`) provee un conjunto en el que se puede insertar, eliminar, y testear pertenencia en tiempo logarítmico (de comparaciones y/o copias).

A diferencia del `ConjuntoLineal`, la operación agregarRápido no permite bajar el costo de agregar un elemento al conjunto.

Las complejidades de las operaciones son las siguientes:

ConjuntoLog

proc	complejidad
`conjVacío`	$O(1)$
`tamaño`	$O(1)$
`pertenece`	$O(\log n)$
`agregar`	$O(\log n)$
`agregarRápido`	$O(\log n)$
`sacar`	$O(\log n)$
`unir`	$O((n+m)\ log(n+m))$
`restar`	$O((n+m)\ log(n+m))$
`intersecar`	$O((n+m)\ log(n+m))$

donde n es la cantidad de elementos del conjunto y m es la cantidad de elementos del segundo conjunto interviniente (si lo hubiera) Si el costo de copiar un elemento o de comparar dos elementos no fuera constante, se debe tener en cuenta en la complejidad de las operaciones.

Es importante destacar que, más allá de la cota de cada operación individual, si se debe recorrer un conjunto por completo, el costo total del recorrido es $O(n)$.

La representación por la que optamos es un árbol AVL. Aquí suponemos T es un tipo básico o, en caso de no serlo (e.g., es una tupla o struct), el orden de los elementos quedará definido sólo por la primera componente.

```
Módulo ConjuntoLog<T> implementa Conjunto⟨T⟩ <
  var elementos: AVL<T>
  var tamaño: int

  // . . .
>
```

5.2.3. ConjuntoDigital

Conjunto implementado con Tries.

El módulo ConjuntoDigital provee un conjunto básico en el que se puede insertar, eliminar, y testear pertenencia en tiempo logarítmico (de comparaciones y/o copias). En cuanto al recorrido de los elementos, se provee un iterador bidireccional.

A diferencia del ConjuntoLineal, la operación agregarRápido no permite bajar el costo de agregar un elemento al conjunto.

Las complejidades de las operaciones son las siguientes:

ConjuntoLog

proc	complejidad
`conjVacío`	$O(1)$
`tamaño`	$O(1)$
`pertenece`	$O(\|e\| * \|a\|)$
`agregar`	$O(\|e\| * \|a\| * eq(a) + cp(e))$
`agregarRápido`	$O(\|e\| * \|a\| * eq(a) + cp(e))$
`sacar`	$O(\|e\| * \|a\| * eq(a))$
`unir`	$O(m * \text{máx}(c2) * \|a\| * eq(a)))$
`restar`	$O(n * m * \|a\|)$
`intersecar`	$O(n * \text{máx}(c1) * m\,\text{máx}(c2))$

donde $|e|$ es la longitud de la clave usada para indexar (i.e., el propio elemento, si fuera simple, o la primera componente, si fuera algún tipo compuesto), n es la cantidad de elementos y máx($c1$) la clave más grande del primer conjunto, y m es la cantidad de elementos y máx($c2$) la clave más grande del segundo conjunto interviniente (si lo hubiera). Para un uso estándar, donde cada símbolo es un carácter que está acotado para algún idioma determinado, el tamaño del alfabeto y el costo de comparación tienen costo constante, por lo que se desestiman en el análisis asintótico. Finalmente, para casos en lo que lo elementos son de longitud acotada, el costo de todas las operaciones es $O(1)$.

La representación por la que optamos es un árbol Trie. Aquí suponemos T es un tipo que tiene un orden parcial. En caso de ser algún tipo definido por una tupla o struct, el orden será definido sólo por la primera componente.

Módulo ConjuntoDigital<T> **implementa** Conjunto$\langle T \rangle$ <

```
  var elems: Trie<T>
  var tamaño: int

  // . . .
>
```

```
Modulo ConjuntoDigital<T> implementa Conjunto<T> {
    var elems: Trie<T>        // los elementos del conjunto.
    var tamaño: int
    ...
}
```

6. Diccionario

6.1. Especificación

TAD Diccionario$\langle K, V \rangle$ {

obs elems : dicc$\langle K, V \rangle$

proc diccionarioVacío() : Diccionario$\langle K, V \rangle$ {

asegura { $res.elems = \langle\rangle$ }

}

```
    proc igualdad(D_1 : Diccionario⟨K, V⟩, D_2 : Diccionario⟨K, V⟩) : bool {
        asegura {
            res = true ↔ |D_1.elems| = |D_2.elems| ∧_L
                (∀k : T) (t ∈ D_1.elems ↔ t ∈ D_2.elems) ∧
                (∀k : T) (t ∈ D_1.elems →_L D_1.elems[t] = D_2.elems[t])
        }
    }

    proc está(in d : Diccionario⟨K, V⟩, in k : K) : bool {
        asegura { res = true ↔ k ∈ d.elems }
    }

    proc definir(inout d : Diccionario⟨K, V⟩, in k : K, in v : V) {
        requiere { d = D_0 }
        asegura { d.elems = setKey(D_0.elems, k, v) }
    }

    proc obtener(in d : Diccionario⟨K, V⟩, in k : K) : V {
        requiere { k ∈ d.elems }
        asegura { res = d.elems[k] }
    }

    proc borrar(inout d : Diccionario⟨K, V⟩, in k : K) {
        requiere { d = D_0 ∧ k ∈ d.elems }
        asegura { d.elems = delKey(D_0.elems, k) }
    }

    proc definirRápido(inout d : Diccionario⟨K, V⟩, in k : K, in v : V) {
        requiere { d = D_0 ∧ k ∉ d.elems }
        asegura { d.elems = setKey(D_0.elems, k, v) }
    }

    proc tamaño(in d : Diccionario⟨K, V⟩) : ℤ {
        asegura { res = |d.elems| }
    }
}
```

6.2. Implementaciones

6.2.1. DiccionarioLineal

El módulo `DiccionarioLineal` (o `DiccionarioLista`) implementa un TAD Diccionario en el que se puede definir, borrar, y verificar si una clave está definida en tiempo lineal. Cuando ya se sabe que la clave a definir no esta definida en el diccionario, la definición se puede hacer en tiempo $O(1)$.

Las complejidades de las operaciones son las siguientes:

DiccionarioLineal

proc	complejidad
`diccionarioVacío`	$O(1)$
`está`	$O(n)$
`definir`	$O(n)$
`definirRápido`	$O(1)$
`obtener`	$O(n)$
`borrar`	$O(n)$
`tamaño`	$O(1)$

donde n es la cantidad de claves definidas en el diccionario. Si el costo de copiar un elemento o de comparar dos elementos no fuera constante, se debe tener en cuenta en la complejidad de las operaciones.

La implementación consiste en dos listas, una de claves y otra de significados. La lista de claves no puede tener repetidos, mientras que la de significados si puede. Ademas, la i-ésima clave de la lista se asocia al i-ésimo significado.

Módulo DiccionarioLineal<K, V> **implementa** Diccionario$\langle K, V \rangle$ <

```
  var claves: ListaEnlazada<K>
  var valores: ListaEnlazada<V>
  var tamaño: int

  // . . .
```

>

6.2.2. DiccionarioLog

El módulo `DiccionarioLog` (o `DiccionarioAVL`) implementa un TAD Diccionario en el que se puede definir, borrar, y verificar si una clave está definida en tiempo logarítmico. A diferencia del `DiccionarioLineal`, no cambia el costo en definirRápido con respecto a definir (i.e., no cambia el costo saber que la clave no está definida).

Las complejidades de las operaciones son las siguientes:

DiccionarioLog

proc	complejidad
`diccionarioVacío`	$O(1)$
`está`	$O(\log n)$
`definir`	$O(\log n)$
`definirRápido`	$O(\log n)$
`obtener`	$O(\log n)$
`borrar`	$O(\log n)$
`tamaño`	$O(1)$

donde n es la cantidad de claves definidas en el diccionario. Si el costo de copiar un elemento o de comparar dos elementos no fuera constante, se debe tener en cuenta en la complejidad de las operaciones.

Como ocurría con ConjuntoLog, si fuera necesario recorrer el diccionario por completo, el costo total del recorrido es $O(n)$.

La implementación es un árbol AVL de tuplas, donde la primera componente es la clave, y la segunda es el valor. Aquí suponemos que K es un tipo básico (e.g., no es tupla ni struct) y tiene un orden parcial.

Módulo ConjuntoLog<K, V> **implementa** Diccionario$\langle K, V \rangle$ <

```
  var definiciones: AVL<Tupla<K, V>>
  var tamaño: int

  // . . .
```

>

6.2.3. DiccionarioDigital

El módulo `DiccionarioDigital` o `DiccionarioTrie` implementa el TAD Diccionario. Provee un diccionario en el que se puede definir, borrar, y verificar si una clave está definida en tiempo lineal con respecto a la longitud de la clave más larga.

Las complejidades de las operaciones son las siguientes:

DiccionarioDigital

proc	complejidad
`diccionarioVacío`	$O(1)$
`está`	$O(\|k\|)$
`definir`	$O(\|k\|)$
`definirRápido`	$O(\|k\|)$
`obtener`	$O(\|k\|)$
`borrar`	$O(\|k\|)$
`tamaño`	$O(1)$

donde $|k|$ es el tamaño de la clave más largo. Es habitual usar este módulo con claves de tamaño acotado, en cuyo caso todos los procedimientos anteriores pasan a tener costo constante ($O(1)$).

La representación por la que optamos es un árbol Trie de tuplas, donde la primera componente es la clave, que debe ser de tipo string o Secuencia, y la segunda es el valor. Aquí suponemos K es un tipo simple (e.g., no es tupla ni struct) y tiene un orden parcial.

Módulo DiccionarioDigital<K, V> **implementa** Diccionario$\langle K, V \rangle$ <

```
  var definiciones: Trie<Tupla<K, V>>
  var tamaño: int

  // . . .
```

>

Apéndice: Detalle de la complejidad de las operaciones sobre vector

Definimos nuestra $f(n)$ como

$$f(n) = \begin{cases} n & \text{si } n = 2^k \text{ para algún } k \\ 1 & \text{en caso contrario} \end{cases}$$

Notar que $\sum_{i=1}^{n} \frac{f(j+i)}{n} \to 1$ cuando $n \to \infty$, para todo $j \in \mathbb{N}$. En otras palabras, la inserción consecutiva de n elementos costará $O(1)$ operaciones por elemento, en términos asintóticos.