Ejercico 11 - Texto

a) Describir la estructura a utilizar, documentando claramente cómo la misma resuelve el problema y cómo cumple con los requerimientos de eficiencia. El diseño debe incluir sólo la estructura de nivel superior. Para justificar los órdenes de complejidad, describa las estructuras soporte. Importante: si alguna de las estructuras utilizadas requiere que sus elementos posean una función especial (por ejemplo, comparación) deberá describirla.

> En particular, asumimos que trabajaremos sólo con textos en español, y por lo tanto podemos dar una cota para la longitud de la palabra más larga que puede aparecer en el texto.

```
(1) Esto nos permite poder usar un Trie para trabajar con las palabras más
eficientemente. Las operaciones tendrían costo constante O(1). Esto se
debe a que insertar o buscar cualquier palabra en un trie depende de su
longitud. Si sabemos que existe una palabra de longitud máxima
(supongamos, 25 letras), cualquier palabra que queramos insertar o buscar
en el trie requerirá a lo sumo (como máximo) 25 operaciones. Por lo tanto,
como independientemente de la palabra vamos a realizar como máximo un
número constante de operaciones, decimos que las operaciones tienen costo
O(1)
```

> subtexto(in t: Texto, in desde: int, in hasta: nat): Texto
>
> Devuelve el texto correspondiente al fragmento de t que comienza en la posición desde y finaliza en la posición hasta.
>
> O(hasta – desde) en el peor caso

Para cumplir esta complejidad deberíamos poder acceder a cada palabra del subtexto en O(1) ya que `hasta - desde = len` y `len*O(1) = O(len)`. Si tuviéramos un arreglo con las palabras del texto leído en el orden en que aparecen, podríamos indexar sin problemas usando `hasta` para leer hasta `desde`.

> cambiarPalabra(inout t: Texto, in vieja: palabra, in nueva: palabra) Cambia todas las ocurrencias en el texto de la palabra vieja por la nueva
>
> O(k) en el peor caso, donde k es la cantidad de veces que se repite la palabra a cambiar

Para cumplir esto pareciera que me conviene tener algo que para cada palabra que aparece en el texto me diga la posición de todas sus apariciones. Además, tenemos que poder encontrar la clave que buscamos en tiempo constante O(1), porque sino la complejidad de cambiarPalabra no dependería solamente de las apariciones de la palabra vieja. Sabiendo además lo mencionado en (1), podemos utilizar un diccionario implementado sobre un Trie cuyas claves son las palabras del texto y los significados un conjunto de posiciones dentro del arreglo planteado en (2) para poder buscarlas. Así, busco cada aparición de la palabra vieja en O(1) y las reemplazo por la nueva también en O(1). Suponiendo que `long(conjApariciones) = k`, la complejidad total será O(k).

Sin embargo, también necesito actualizar las posiciones en que aparece la palabra nueva porque podría pasar que

- No estuviera previamente: en ese caso tengo que sumarla al diccionario de palabras la nueva y asignarle las mismas posiciones que la anterior.
- Si estuviera previamente: tengo que realizar la unión de las posiciones de la palabra vieja y las posiciones que tenía previamente la nueva y ponerlo en posiciones de la nueva. Además, remover la vieja del diccionario.

> palabrasMasRepetidas(in t: text) → conj(palabra)
>
> Devuelve el conjunto de palabras que más se repiten en el texto
>
> O(1) en el peor caso. Puede generar aliasing

Primero observamos que nos dice que puede generar aliasing, eso lo podemos tomar como una "pista" para saber que podemos devolver referencias a cosas. Por otro lado, al tener que devolverse en O(1) vamos a tener que tener "precalculado" el conjunto de palabras. Para eso, además, vamos a tener que ir completándolo a medida que ingresamos el texto e ir teniéndolo actualizado si obtenemos una nueva palabra más repetida que la palabra más repetida hasta el momento.

Por lo tanto, quizás resulte útil tener un contador que indique la mayor cantidad de repeticiones registrada hasta el momento en el texto. Así, al ingresar una palabra puedo saber si su cantidad de repeticiones supera a la más repetida actualmente ya que eso sería la longitud del conjunto que indica las posiciones actuales (obtenible en O(1) gracias al dicc del item anterior) más uno (por la nueva introducción). En ese caso, desecho el conjunto de palabras que tengo hasta el momento e introduzco uno que sólo contenga la palabra que acaba de superar el valor previo. Luego, cada vez que una palabra sea agregada, si tiene el mismo valor, se la introduce también al conjunto de palabras más repetidas (O(1)).

Al cambiarPalabra voy a realizar la misma verificación que hice antes al agregar. La nueva palabra, dado que la cambié a partir de la anterior, ¿tiene la misma cantidad de repeticiones que el máximo o más? Esto se hace contando las posibles repeticiones de la palabra nueva previas a cambiar + la cantidad que tenía la palabra anterior (pero que ahora es la nueva). Si es así, tendría que poder actualizar el contador, el conjunto de las más repetidas y las posiciones de las palabras sin que se vea afectada la complejidad de cambiarPalabra. Para eso, tengo que poder acceder en O(1) a todo. Para eso el conjunto de las más repetidas debería ser un ConjuntoTrie también.

- Si es mayor al max: creo un nuevo conjunto con esa palabra nada más y actualizo max. Además, como dijimos antes, hago la unión del conjunto de posiciones de anterior y del nuevo (si tuviera posiciones previamente) para reemplazar en posiciones.
- Si es igual al max: tengo que reemplazar la palabra anterior por la nueva y no modificar el max. Además, hacer la unión del conjunto de posiciones como antes.
- Si es menor al max: sólo tengo que hacer la unión de posiciones.

ESTRUCTURA

```
Palabra es string

Modulo TextoImpl implementa Texto {
```

```
    var palabras:            Vector(Palabra),
    var posiciones:       DiccionarioTrie(Palabra, ConjuntoLineal(int)),
    var repeticionMaxima: int,
    var lasMasRepetidas:  ConjuntoTrie(Palabra),
}
```

Sea `ti` una instancia del `TextoImpl`, se tiene que:

- `ti.palabras`: es un arreglo que contiene todas las palabras del texto en el mismo orden en que son agregadas
- `ti.posiciones`: es un diccionario implementado sobre Trie cuyas claves son todas las palabras que aparecen en ti.palabras y cada significado asociado es un conjunto de las posiciones en que aparece esa palabra (la clave) en ti.palabras.
- `ti.repeticionMaxima`: es un natural que representa el número de repeticiones que tienen las palabras más repetidas de ti.palabras.
- `ti.lasMasRepetidas`: es un conjunto implementado sobre Trie que contiene las palabras de ti.palabras que aparecen ti.repeticionMaxima veces.

FUNCIÓN DE ABSTRACCIÓN (b)

```
pred Abs(ti: TextoImpl, t: Texto) {
    1) ∧ 2)
}
```

donde:

- ti.palabras tiene la misma cantidad de palabras que el texto representado
    1. `|t.palabras| == ti.palabras.Longitud()`
- las palabras de ti.palabras están en el mismo orden y son las mismas
    2. `forall i: int :: 0 <= i < |t.palabras| ==>L t.palabras[i] = ti.palabras.Obtener(i)`

INVARIANTE DE REPRESENTACIÓN (b)

```
pred Rep(ti: TextoImpl) {
    1) ∧ 2) ∧ 3.a) ∧ 3.b) ∧ 4) ∧ 5) ∧ 6) ∧ 7) ∧L 8) ∧ 9)
}
```

- El número de repeticiones máximas es cero (al crear el texto vacío) o positivo y como mucho puede ser la cantidad total de palabras en el texto (Nota: esta afirmación se desprende también de la 5)
    1. `0 <= ti.repeticionMaxima <= ti.palabras.Longitud()`
- No hay superposición de posiciones entre las distintas claves de ti.posiciones (Nota: esta afirmación se desprende de la 3 y 4)
    2. `forall p: Palabra :: ti.posiciones.Esta(p) ==>L (forall p': Palabra :: ti.posiciones.Esta(p') && p != p' ==>L (forall i: int ::`

`ti.posiciones.Obtener(p).Esta(i) <==> ¬ti.posiciones.Obtener(p').Esta(i)))` or

3. `forall p: Palabra :: ti.posiciones.Esta(p) ==>L (forall p': Palabra :: ti.posiciones.Esta(p') && p != p' ==>L (forall i: int :: 0 <= i < ti.palabras.Longitud() ==>L (ti.posiciones.Obtener(p).Esta(i) ==>L ¬ti.posiciones.Obtener(p').Esta(i))))`

- Toda clave de ti.posiciones es clave sii está en ti.palabras 3.a) `forall p: Palabra :: ti.posiciones.Esta(p) ==>L exists i: int :: 0 <= i < ti.palabras.Longitud() &&L p == ti.palabras.Obtener(i)` 3.b) `forall i: int :: 0 <= i < ti.palabras.Longitud() ==>L ti.palabras.Obtener(i) in ti.posiciones`
- Toda posición del significado de ti.posiciones es una posición de ti.palabras que contiene la palabra de la clave

4. `forall p: Palabra :: ti.posiciones.Esta(p) ==>L ti.palabras.Obtener(ti.posiciones.Obtener(p)) == p`

- Si una palabra p aparece en la posición i en el texto, debería entonces ti.posiciones contener i en el significado de p dentro de ti.posiciones

5. `forall i: int :: 0 <= i < ti.palabras.Longitud() ==>L (exists p: Palabra :: ti.palabras.Obtener(i) == p &&L (ti.posiciones.Esta(p) &&L ti.posiciones.Obtener(p).Pertenece(i)))`

- ti.repeticionMaxima es igual al valor de la mayor cantidad de repeticiones de ti.posiciones, representado como la longitud del significado de cada clave de ti.posiciones

6. `exists p: Palabra :: ti.posiciones.Esta(p) ==>L ti.posiciones.Obtener(p).Longitud() == ti.repeticionMaxima && (forall p': Palabra :: ti.posiciones.Esta(p') ==>L ti.posiciones.Obtener(p').Longitud() <= ti.repeticionMaxima)`

- ti.lasMasRepetidas es un subconjunto de las claves de ti.posiciones

7. `forall p: Palabra :: ti.lasMasRepetidas.Pertenece(p) ==>L ti.posiciones.Esta(p)`

- Toda clave de ti.lasMasRepetidas tiene una cantidad de apariciones (obtenida del significado de ti.posiciones) igual a ti.repeticionMaxima

8. `forall p: Palabra :: ti.lasMasRepetidas.Pertenece(p) ==>L ti.posiciones.Obtener(p).Longitud() == ti.repeticionMaxima` (Nota: puedo pedir directamente el significado en posiciones porque en 7 ya dije que si está en lasMasRepetidas entonces está en posiciones pero debo conectarlo con un yLuego)

- Si una palabra tiene las máximas repeticiones entonces está en el conjunto de las más repetidas

9. `forall p: Palabra :: ti.posiciones.Esta(p) ==>L ((ti.palabras.Obtener(ti.posiciones.Obtener(p)) == ti.repeticionMaxima) ==> ti.lasMasRepetidas.Pertenece(p))`

ALGORITMOS (c, d y e)

> Nota: Como ya en la estructura determiné de qué tipo son cada una de mis variables no es necesario poner el nombre del módulo adelante de cada función como se vio en alguna clase previa o dice el apunte. Si no hubiera hecho eso en la estructura, sería necesario decir `DiccionarioTrie.DiccionarioVacio()` para poder asignarlo a res.palabras, por dar un ejemplo.

> Nota2: Recuerden que las complejidades en un Trie de string sería O(|l|) donde l es la palabra que estoy queriendo buscar, insertar o borrar. Como en nuestro caso existe una L que es la palabra más larga y todas las palabras tienen menor o igual longitud que esa palabra podemos considerar que la complejidad no depende de la palabra que estemos queriendo buscar, insertar o borrar sino que siempre vamos a hacer como máximo |L| operaciones y podemos considerar esa complejidad como constante, es decir, O(1).

```
proc nuevoTexto(): Texto
    Complejidad: O(1)
{
    res.texto                := VectorVacío()      // O(1)
    res.palabras             := DiccionarioVacío() // O(1)
    res.repeticionMaxima     := 0                  // O(1)
    res.lasMasRepetidas      := ConjVacio()        // O(1)
}
```

```
proc agregarPalabra(inout t: Texto, in p: Palabra)
    Complejidad: O(n)
{
    // Agrego la palabra en el texto propiamente
    t.texto.AgregarAtrás(p)                        // O(n) para ser
exactos, pero en realidad se lo puede considerar O(1) amortizado como se
explicó en clase

    // Actualizo la estructura restante
    ActualizarTexto(t, p, t.texto.Longitud()-1)    // O(1)
}
```

```
(c) proc cambiarPalabra(inout t: Texto, in vieja: Palabra, in nueva:
Palabra)
    Complejidad: O(k), donde k es la cantidad de apariciones de la palabra
vieja
{
    // Actualizo el vector que representa al texto
    indices := t.palabras(vieja)                   // O(1)
    for i in indices {                             // O(k)
        t.texto.ModificarPosición(i, nueva)        // O(1)
    }

    // Actualizo el resto de la estructura
    for i in nuevosIndices {                       // O(k)
        ActualizarTexto(t, nueva, i)               // O(1)
    }

    // Elimino las apariciones de vieja
    t.palabras.Borrar(vieja)                       // O(1)
    if t.lasMasRepetidas.Esta(vieja) then          // O(1)
```

```
        t.lasMasRepetidas.Borrar(vieja)             // O(1)
}
```

```
proc posiciones(in t: Texto, in p: Palabra): Conjunto<int>
    Complejidad: O(n)
{
    return copy(t.palabras.Obtener(p))              // O(n)
}
```

```
(b) proc subtexto(in t: Texto, in desde: int, in hasta: int): Texto
    Complejidad: O(hasta - desde)
{
    txt := NuevoTexto()                             // O(1)

    // Leo y construyo el nuevo vector de palabras
    txt.texto := VectorVacío()                      // O(1)
    for i := desde; i < hasta; i++ {                // O(hasta - desde)
        txt.texto.AgregarAtrás(t.texto.Obtener(i))  // O(1)
    }

    // Actualizo el resto de la estructura de acuerdo
    // al subtexto seleccionado
    for i := 0; i < hasta-desde; i++ {              // O(hasta - desde)
        ActualizarTexto(txt, txt.texto[i], i)       // O(1)
    }
}
```

```
proc ActualizarTexto(inout t: Texto, in p: Palabra, in i: int)
    Complejidad: O(1)
    Requiere:    {0 <= i < t.texto.Longitud() &&L p == t.texto.Obtener(i)}
{
    // Agrego la aparición de la palabra
    if t.palabras.Está(p) then                       // O(1)
        posicPrevias := t.palabras.Obtener(p)        // O(1)
        nuevasPosic := posicPrevias.AgregarRápido(p) // O(1)
        t.palabras.Definir(p, nuevasPosic)           // O(1)
    else
        nuevasPosic := ConjuntoVacío()               // O(1)
        nuevasPosic := nuevasPosic.AgregarRápido(p)  // O(1)
        t.palabras.Definir(p, nuevasPosic)           // O(1)

    // Me fijo si tengo que actualizar el número de repeticiones máximas
    // Si la nueva palabra tiene más repeticiones que la más repetida
    repeticionesDeP := t.palabras.Obtener(p).Tamaño()   // O(1)
    if repeticionesDeP > t.repeticionMaxima then        // O(1)
        t.repeticionMaxima := repeticionesDeP           // O(1)
```

```
        t.lasMasRepetidas := ConjuntoVacío()            // O(1)
        t.lasMasRepetidas.AgregarRápido(p)              // O(1)

    // Si la nueva palabra tiene las mismas repeticiones que la más
repetida
    if repeticionesDeP == t.repeticionMaxima then       // O(1)
        t.lasMasRepetidas.AgregarRápido(p)              // O(1)

    // Si la nueva palabra tiene menos que la más repetida, no hago nada
}
```

```
(d) proc masRepetidas(in t: Texto): Conjunto<Palabra>
    Complejidad: O(1)
{
    return t.lasMasRepetidas                            // O(1)
}
```