Tries

Algoritmos y Estructuras de datos

Alejandro Schulz - 2° Cuatri 2024

Mini repaso

- Qué es un Trie?

Mini repaso

- Qué es un Trie?
  - Es un árbol k+1-ario para alfabetos de k elementos.

- Qué representa cada Nodo?

Mini repaso

- Qué es un Trie?
  - Es un árbol k+1-ario para alfabetos de k elementos.

- Qué representa cada Nodo?
  - Cada nodo del árbol representa una secuencia de elementos del alfabeto.

- Cómo nos movemos de Nodo a Nodo?

Mini repaso

- Qué es un Trie?
  - Es un árbol k+1-ario para alfabetos de k elementos.

- Qué representa cada Nodo?
  - Cada nodo del árbol representa una secuencia de elementos del alfabeto.

- Cómo nos movemos de Nodo a Nodo?
  - Si las claves son strings, “extendemos” la secuencia agregando un carácter.
  - Si las claves son enteros, lo hacemos con dígitos o bits.

Diseño con Tries

Ejercicio 10. Implemente un Trie utilizando arreglos y listas enlazadas para los nodos.

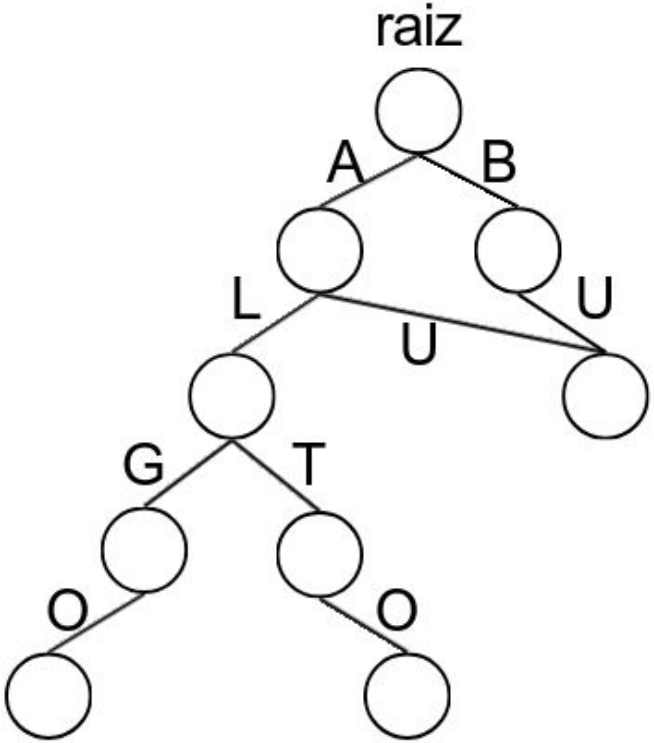
- Describa en castellano el invariante de representación
- Escriba los algoritmos para las operaciones `buscar` y `agregar` y justifique la complejidad de cada operación.
- ¿Qué diferencias observa entre ambas implementaciones? ¿Qué ventajas y desventajas tiene cada una? En qué casos utilizaría cada una?

Invariante de Representación

Para saber que tiene que pedir el inv de rep pensemos qué cumple un Trie bien hecho y qué no cumple o dónde se rompe uno mal armado.
El invRep, en líneas generales, cambia mucho si hablamos de arreglos o de listas enlazadas?

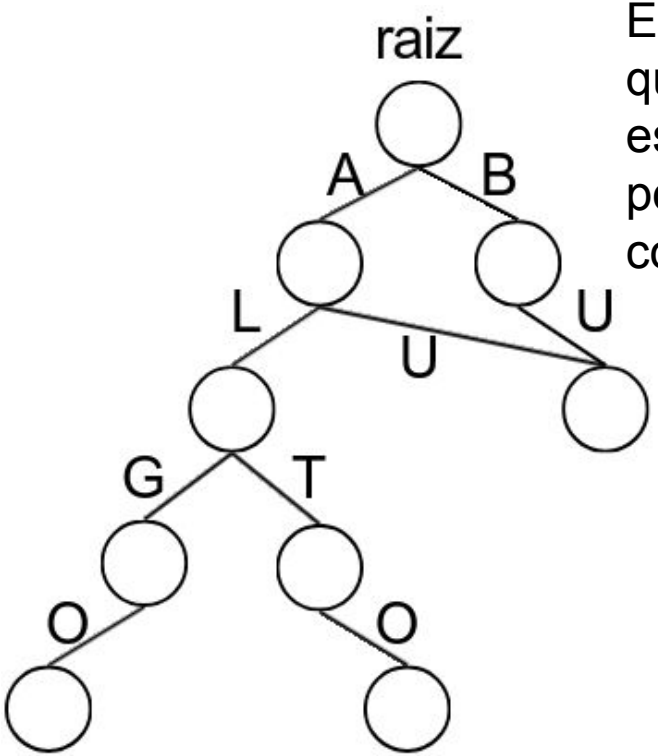
Invariante de Representación

Son tries válidos?



Invariante de Representación

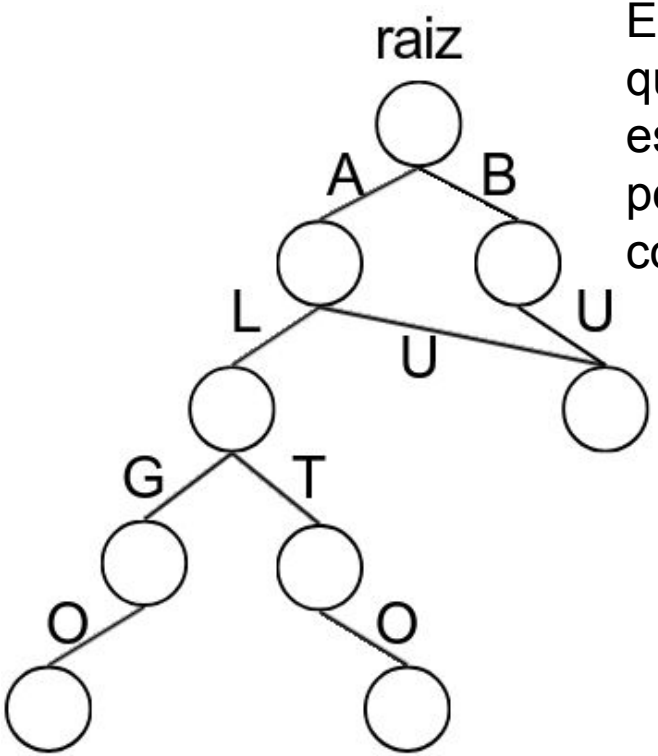
Son tries válidos?



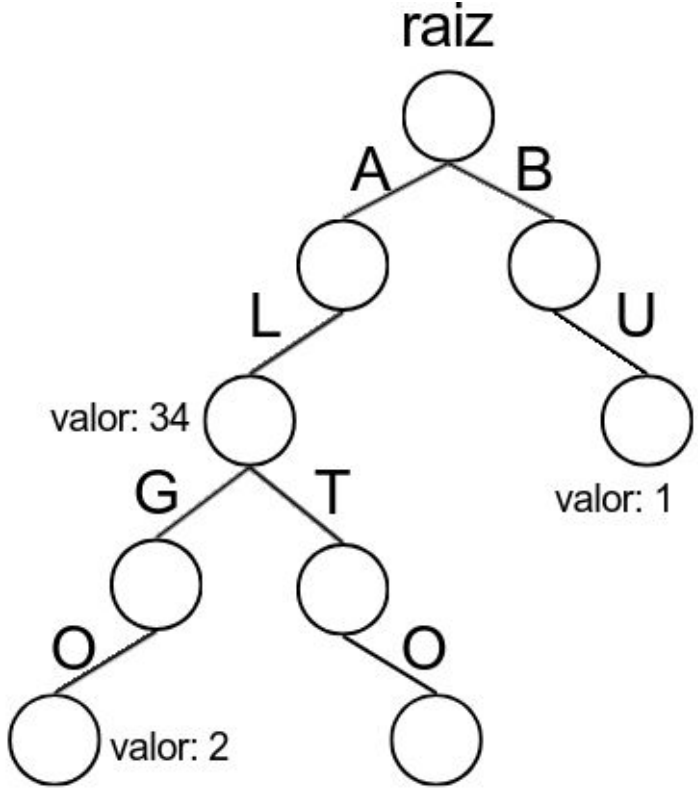
El invRep nos tendría que decir que este NO es un trie válido porque tiene un nodo con dos padres.

Invariante de Representación

Son tries válidos?

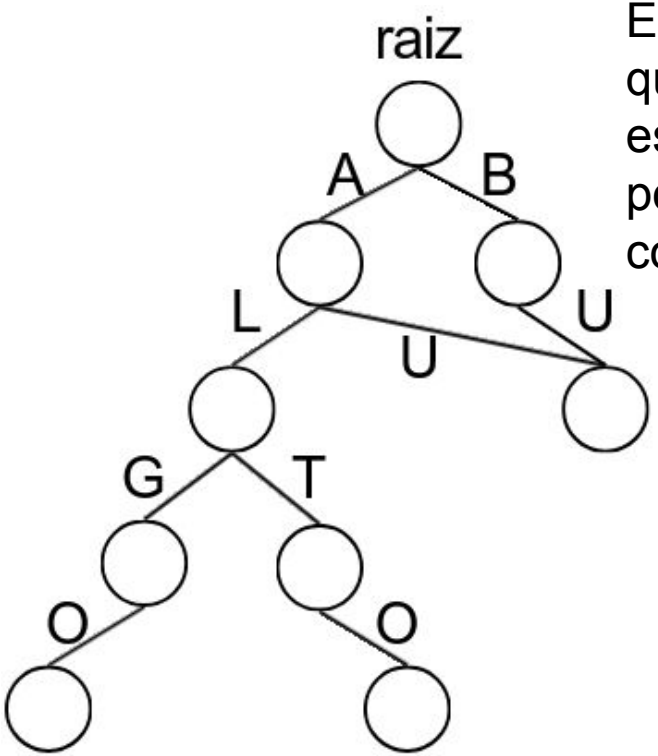


El invRep nos tendría que decir que este NO es un trie válido porque tiene un nodo con dos padres.

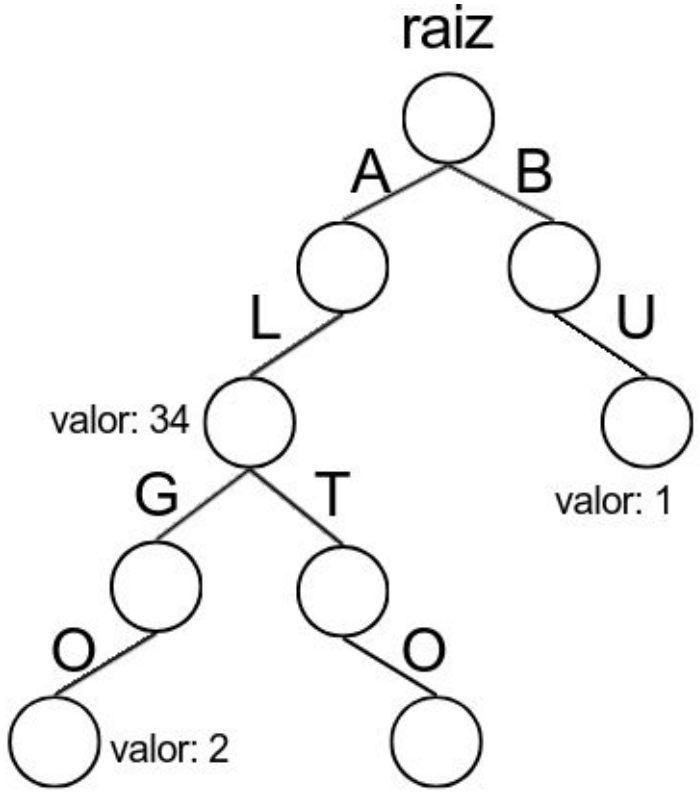


Invariante de Representación

Son tries válidos?



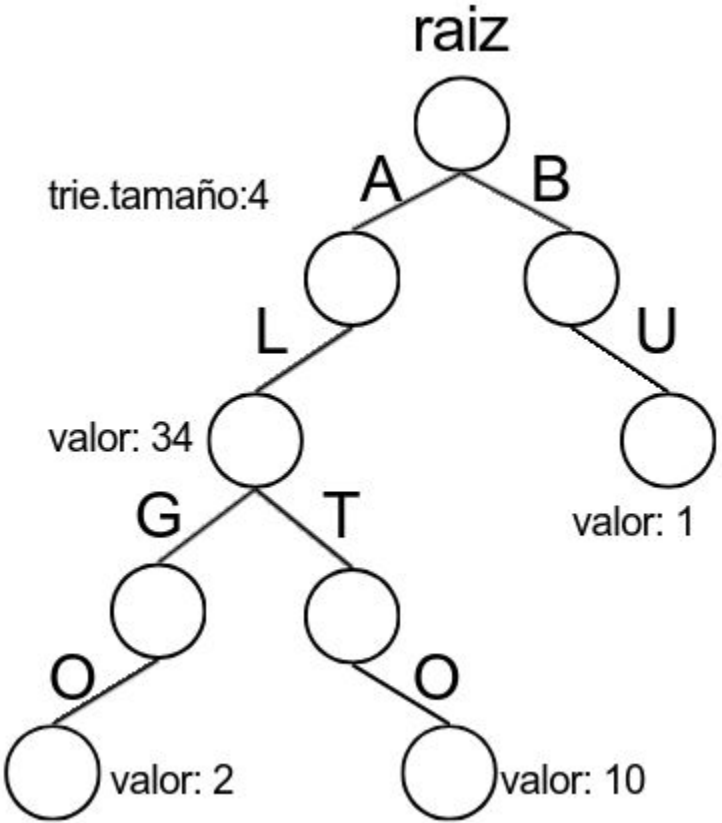
El invRep nos tendría que decir que este NO es un trie válido porque tiene un nodo con dos padres.



Depende de que hagamos en Eliminar().

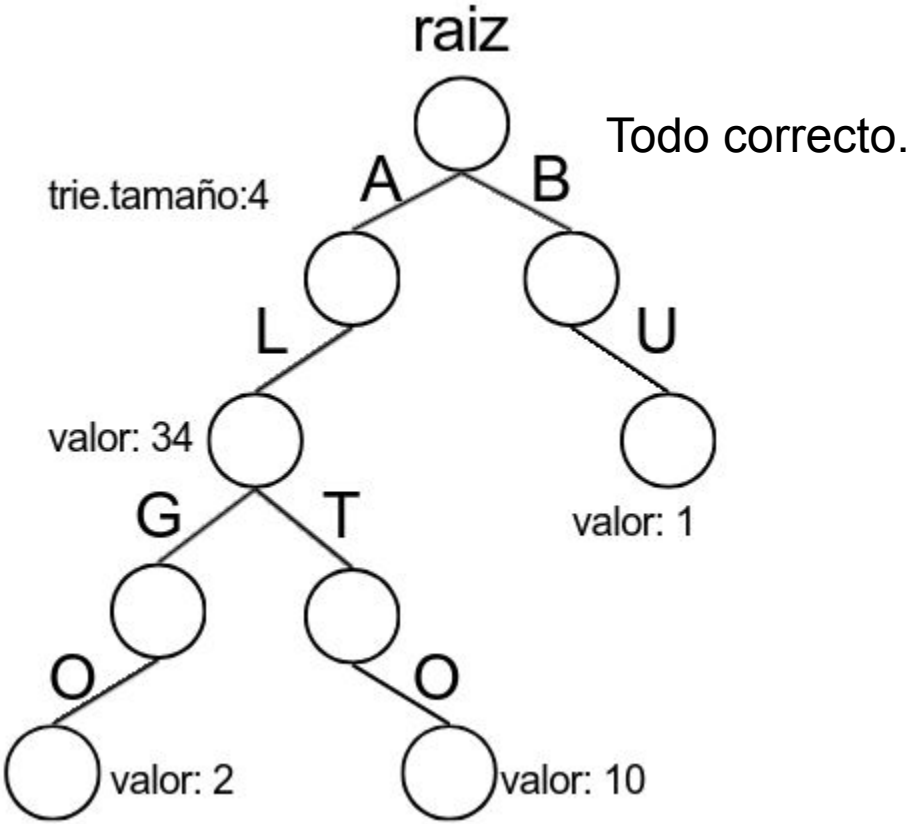
Invariante de Representación

Son tries válidos?



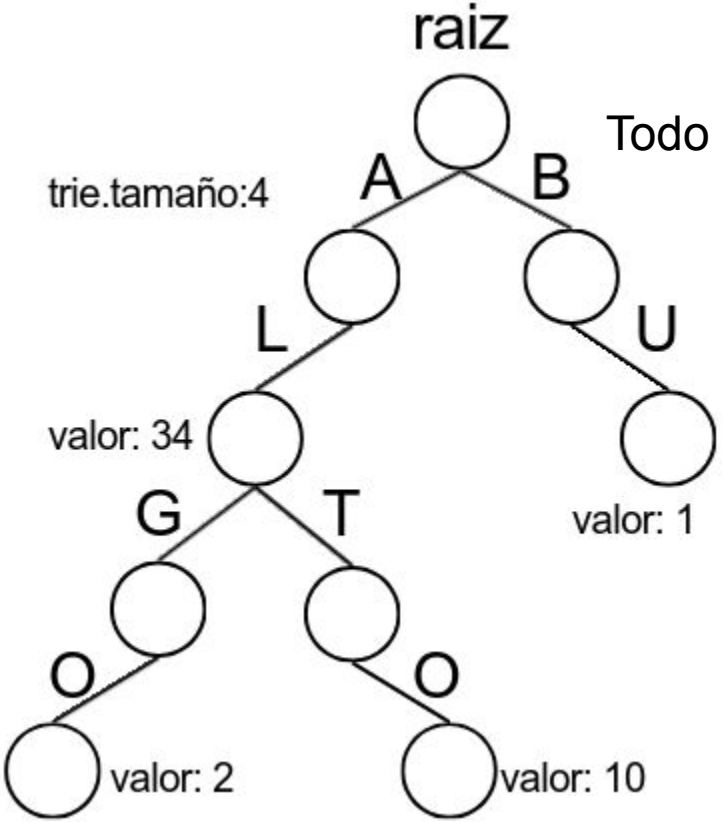
Invariante de Representación

Son tries válidos?

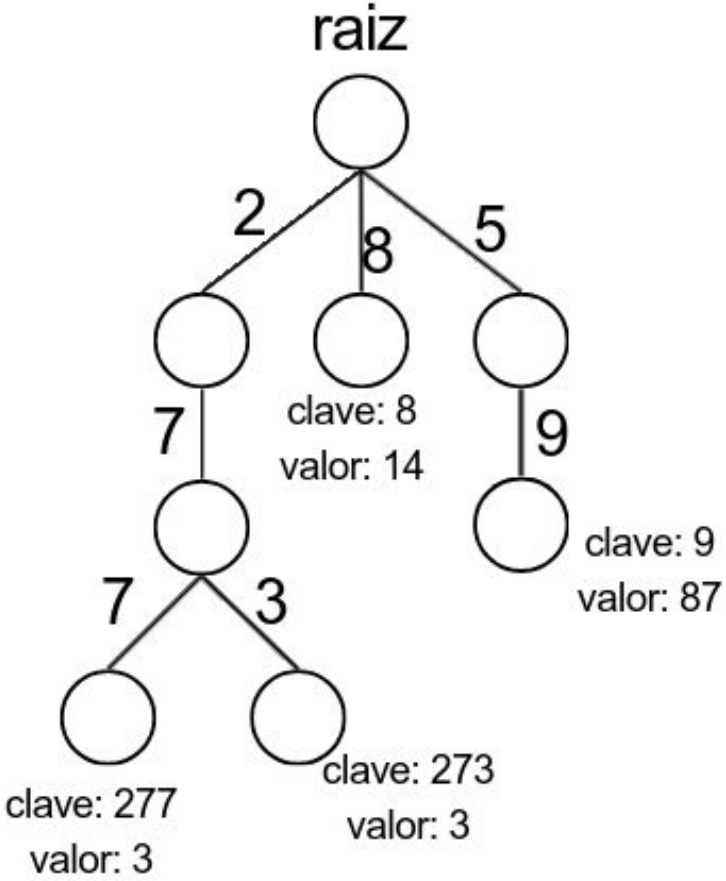


Todo correcto.

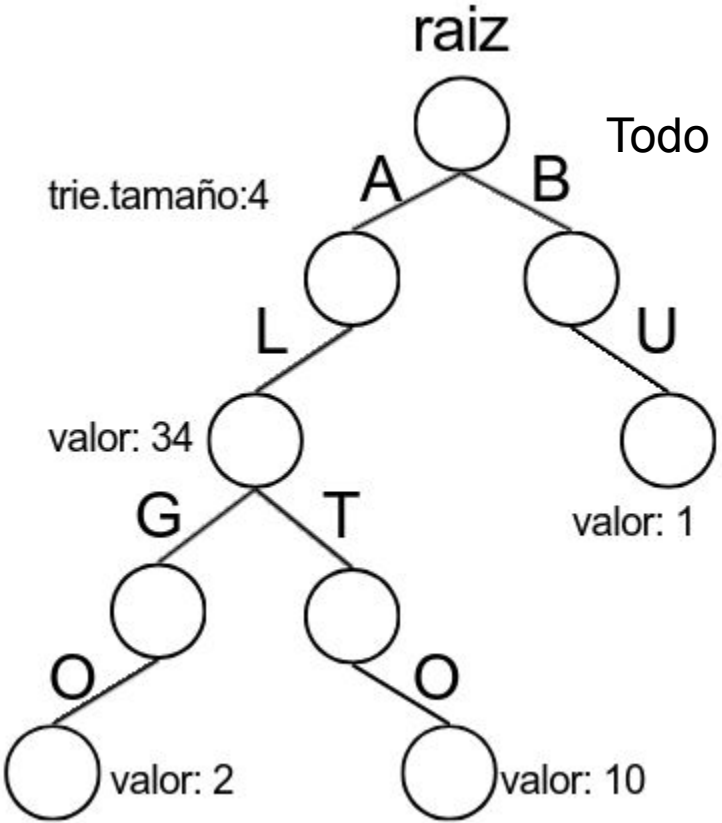
Invariante de Representación



Todo correcto.

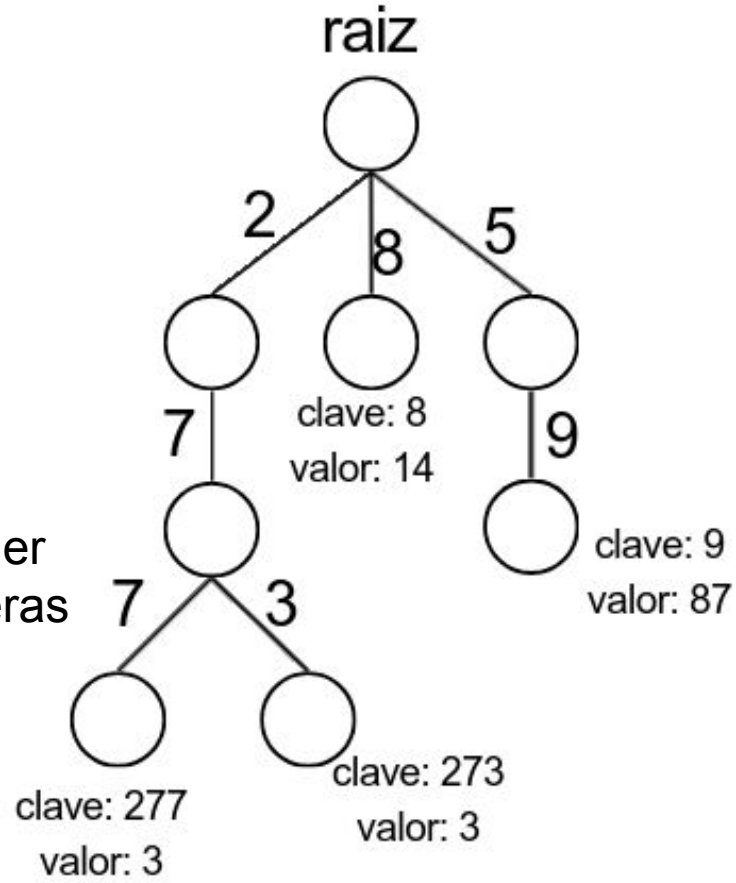


Invariante de Representación



Todo correcto.

Si vamos a tener las claves enteras en los nodos, tienen que corresponder con el camino de la raíz a ese nodo.



Invariante de Representación

Entonces resumiendo:

Invariante de Representación

Entonces resumiendo:

- Cada nodo tiene un único padre, o es la raíz (que no tiene padre).

Invariante de Representación

Entonces resumiendo:

- Cada nodo tiene un único padre, o es la raíz (que no tiene padre).

- Las hojas tienen valores definidos y válidos.
  (entonces Eliminar() borraría los nodos sin hijos y sin valor)
  O podemos no decir nada sobre esto.
  (y que Eliminar() solo saque el valor pero nunca los nodos sin hijos.
  Pensemos que si vamos a usar el Trie para poner y sacar muchas claves estamos dejando muchos nodos inútiles que cuestan mucha memoria.)

Invariante de Representación

Entonces resumiendo:

- Cada nodo tiene un único padre, o es la raíz (que no tiene padre).

- Las hojas tienen valores definidos y válidos.
  (entonces Eliminar() borraría los nodos sin hijos y sin valor)
  O podemos no decir nada sobre esto.
  (y que Eliminar() solo saque el valor pero nunca los nodos sin hijos.
  Pensemos que si vamos a usar el Trie para poner y sacar muchas claves estamos dejando muchos nodos inútiles que cuestan mucha memoria.)

- Si el trie tiene trie.tamaño que pasa? Cant de claves definidas = tamaño.

Invariante de Representación

Entonces resumiendo:

- Cada nodo tiene un único padre, o es la raíz (que no tiene padre).

- Las hojas tienen valores definidos y válidos.
  (entonces Eliminar() borraría los nodos sin hijos y sin valor)
  O podemos no decir nada sobre esto.
  (y que Eliminar() solo saque el valor pero nunca los nodos sin hijos.
  Pensemos que si vamos a usar el Trie para poner y sacar muchas claves estamos dejando muchos nodos inútiles que cuestan mucha memoria.)

- Si el trie tiene trie.tamaño que pasa? Cant de claves definidas = tamaño.

- Si ponemos la clave entera en el mismo nodo donde ponemos su valor entonces el recorrido desde la raíz a este nodo = Nodo.ClaveEntera

Trie con Arreglo

- Escriba los algoritmos para las operaciones `buscar` y `agregar` y justifique la complejidad de cada operación.

Trie con Arreglo

- Escriba los algoritmos para las operaciones `buscar` y `agregar` y justifique la complejidad de cada operación.

Queremos implementar operaciones de trie pero no tenemos un Tad o la estructura del módulo, asumamos algo razonable:

Trie con Arreglo

- Escriba los algoritmos para las operaciones `buscar` y `agregar` y justifique la complejidad de cada operación.

Queremos implementar operaciones de trie pero no tenemos un Tad o la estructura del módulo, asumamos algo razonable:

```
Módulo Trie<T> implementa Diccionario<K,T> {
    var raiz: Nodo<T>
    var tamaño: int
}
```

Trie con Arreglo (vector)

- Escriba los algoritmos para las operaciones **buscar** y **agregar** y justifique la complejidad de cada operación.

Queremos implementar operaciones de trie pero no tenemos un Tad o la estructura del módulo, asumamos algo razonable:

```
Módulo Trie<T> implementa Diccionario<K,T> {
    var raiz: Nodo<T>
    var tamaño: int
}
```

Dónde Nodo en este caso es:

```
Nodo<T> es struct<
    valor: T,
    alfabeto: vector<Nodo>
>
```

//Asumimos que nodo.alfabeto se inicializa todo en null.
//Tambien que new Nodo.valor es invalido.

Trie con Arreglo (vector)

- Escriba los algoritmos para las operaciones `buscar` y `agregar` y justifique la complejidad de cada operación.

Aclaración: Un trie se puede usar para implementar Conjuntos o Diccionarios, para implementar uno u otro cambiamos el nombre de los procs.

Para Conjuntos	Buscar ⟷ Pertenece Agregar ⟷ Agregar
Para Diccionarios	Buscar ⟷ Obtener Agregar ⟷ Definir

Para el ejercicio usamos los nombres que nos piden ya que no especifica qué Tad quiere modelar.

Trie con Arreglo (vector)

Tarea: Modificar todos los procs para que las claves sean números naturales.

La palabra ya tendría que estar definido en el Trie.
Asumimos que palabra != []
Para simplificar asumimos que 0 <= palabra[i] <= 26

proc **Buscar**(in trie: Trie<T>, in palabra: vector<char>) T {

Trie con Arreglo (vector)

Tarea: Modificar todos los procs para que las claves sean números naturales.

La palabra ya tendría que estar definido en el Trie.
Asumimos que palabra != []
Para simplificar asumimos que 0 <= palabra[i] <= 26

```
proc Buscar(in trie: Trie<T>, in palabra: vector<char>) T {
    var i = 0
    var actual = trie.raiz
    while(i < longitud(palabra))
        var letra = palabra[i]
        actual = actual.alfabeto[letra]
        i++
    end while
    var res = actual.valor
    return res
}
```

Trie con Arreglo (vector)

Complejidad de Buscar:

En el peor caso estamos buscando una palabra con k letras.
Cada letra es un nodo que bajamos en el trie,
encontrar el siguiente nodo es $O(1)$,
y tenemos que bajar k letras, así que es $O(k)$.

Trie con Arreglo (vector)

Complejidad de Buscar:

En el peor caso estamos buscando una palabra con k letras.
Cada letra es un nodo que bajamos en el trie,
encontrar el siguiente nodo es $O(1)$,
y tenemos que bajar k letras, así que es $O(k)$.

Si el tamaño de las claves está acotado (k es como máximo un número m) entonces para cualquier clave que busquemos como máximo hacemos $O(m)$ operaciones. **m es constante… entonces O(m) = O(1).**

Sl el tamaño de las claves este acotado es O(1).

Trie con Arreglo (vector)

Para Agregar podemos hacer lo mismo para recorrer el Trie.
Ahora puede que no exista el nodo de la letra siguiente, vamos a tener que crear el nodo y ponerlo en su lugar.
Cuando llegamos al final, definimos el valor.

Trie con Arreglo (vector)

proc **Agregar**(in trie: Trie<T>, in palabra: vector<char>, in valor: T)

Trie con Arreglo (vector)

```
proc Agregar(in trie: Trie<T>, in palabra: vector<char>, in valor: T)
    var i = 0
    var actual = trie.raiz
    while(i < longitud(palabra))
        var letra = palabra[i]
        var siguiente = actual.alfabeto[letra]
        if (siguiente = null) {
            var siguiente= new Nodo<T>
            actual.alfabeto[letra] = siguiente }
        actual = siguiente
        i++
    end while
    if (actual.valor == null) // o la manera de decir que T es invalido
        actual.valor = valor
        trie.tamaño++
```

Trie con Arreglo (vector)

Complejidad de Agregar:

Hacemos algo muy distinto que en Buscar?

Trie con Arreglo (vector)

Complejidad de Agregar:

Hacemos algo muy distinto que en Buscar? No.
Si vamos a agregar una palabra de k letras, recorremos o creamos k nodos y definimos el valor, entonces también es $O(k)$…
Pero cuánto cuesta crear un nodo?

Trie con Arreglo (vector)

Complejidad de Agregar:

Hacemos algo muy distinto que en Buscar? No.
Si vamos a agregar una palabra de k letras, recorremos o creamos k nodos y definimos el valor, entonces también es $O(k)$…
Pero cuánto cuesta crear un nodo?
Creamos un vector con el tamaño del alfabeto, eso es $O(|A|)$.
En el peor caso agregamos una palabra con letras que no estaban.
La complejidad de crear k nodos es $O(kx|A|)$??

Trie con Arreglo (vector)

Complejidad de Agregar:

Hacemos algo muy distinto que en Buscar? No.
Si vamos a agregar una palabra de k letras, recorremos o creamos k nodos y definimos el valor, entonces también es $O(k)$…
Pero cuánto cuesta crear un nodo?
Creamos un vector con el tamaño del alfabeto, eso es $O(|A|)$.
En el peor caso agregamos una palabra con letras que no estaban.
La complejidad de crear k nodos es $O(k \times |A|)$??
No, **el tamaño del alfabeto está acotado** y siempre es constante para letras (27 letras) y para números (10 dígitos).
$O(|A|) = O(1)$ ----> $O(k \times |A|) =$ **$O(k)$**
Si el tamaño de la palabra está acotado entonces **$O(k) = O(1)$.**

Trie con Lista Enlazada

Implementar un trie con listas enlazadas significa que ahora en los nodos en vez de tener un vector<> tenemos ListaEnlazada<>.

Nos alcanza con poner ListaEnlazada<Nodo> como cuando lo hicimos con vector?

Trie con Lista Enlazada

Implementar un trie con listas enlazadas significa que ahora en los nodos en vez de tener un vector<> tenemos ListaEnlazada<>.

Nos alcanza con poner ListaEnlazada<Nodo> como cuando lo hicimos con vector? Nop.

Antes sabíamos el nodo de una letra por su posición en el vector.
Ahora la lista enlazada puede tener o no una letra y en el orden que sea, entonces como sabemos el nodo de una letra?

Trie con Lista Enlazada

Implementar un trie con listas enlazadas significa que ahora en los nodos en vez de tener un vector<> tenemos ListaEnlazada<>.

Nos alcanza con poner ListaEnlazada<Nodo> como cuando lo hicimos con vector? Nop.

Antes sabíamos el nodo de una letra por su posición en el vector.
Ahora la lista enlazada puede tener o no una letra y en el orden que sea, entonces como sabemos el nodo de una letra?
Podemos poner el nodo y la letra que representa juntos.
Una manera es ponerla en nodo.valor junto con el valor de la palabra en el trie. Cómo?

Trie con Lista Enlazada

Implementar un trie con listas enlazadas significa que ahora en los nodos en vez de tener un vector<> tenemos ListaEnlazada<>.

Cambiamos esto:

```
Nodo<LetraValor<T>> es struct<
    valor: LetraValor<T>,
    alfabeto: ListaEnlazada<Nodo<LetraValor<T>>
>
```

donde LetraValor es:

```
LetraValor<T> es struct<
    letra: char
    val: T
>
```

Trie con Lista Enlazada

proc **Buscar**(in trie: Trie<T>, in palabra: vector<char>) T {

Trie con Lista Enlazada

```
proc Buscar(in trie: Trie<T>, in palabra: vector<char>) T {
    var i = 0
    var actual = trie.raiz
    while(i < longitud(palabra))
        var letra = palabra[i]
        actual = NodoDeLetra(actual.alfabeto, letra)
        i++
    end while
    var res = actual.valor.val
    return res
}
```

Trie con Lista Enlazada

```
proc NodoDeLetra(in lista: ListaEnlazada<Nodo<LetraValor<T>>,
in letra: char) Nodo {
        var it = lista.iterador()
        while (it.HaySiguente() && it.valor.letra != letra)
                it.Siguiente()
        end while
        return it
}
```

Trie con Lista Enlazada

Complejidad de Buscar:

Como con el vector, recorrer de nodo a nodo las letras de la palabra es $O(k)$ pero ahora en cada uno hay que recorrer su lista para encontrar el siguiente nodo.
En el peor caso la letra está al final de la lista por lo que hay que recorrerla entera en $\mathrm{NodoDeLetra}()$, y la lista, como el vector, puede tener $O(|A|)$ entradas.
Entonces recorrer k veces listas de tamaño $|A|$ es $O(k_x|A|)$.

Si el tamaño de la clave está acotado y el tamaño del alfabeto también, Buscar es $O(1)$.

Trie con Lista Enlazada

proc **Agregar**(in trie: Trie<T>, in palabra: vector<char>, in valor: T) {

Trie con Lista Enlazada

```
proc Agregar(in trie: Trie<T>, in palabra: vector<char>, in valor: T) {
    var i = 0
    var actual = trie.raiz
    while(i < longitud(palabra))
        var letra = palabra[i]
        actual = AgregarNodoDeLetra(actual.alfabeto, letra)
        i++
    end while
    if (actual.valor.val == null) // o la manera de decir que T es invalido
        actual.valor.val = valor
        trie.tamaño++
}
```

Trie con Lista Enlazada

proc **AgregarNodoDeLetra**(in lista:
ListaEnlazada<Nodo<LetraValor<T>>, in letra: char) Nodo {

Trie con Lista Enlazada

```
proc AgregarNodoDeLetra(in lista:
ListaEnlazada<Nodo<LetraValor<T>>, in letra: char) Nodo {
    var it = lista.iterador()
    while (it.HaySiguente() && it.valor.letra != letra)
        it.Siguiente()
    end while
    if (lista.vacia() || it.valor.letra != letra) {
        var nuevo = new Nodo<LetraValor<T>>
        nuevo.valor.letra = letra
        lista.AgregarAtras(nuevo)
        it = nuevo
    }
    return it
}
```

Trie con Lista Enlazada

Complejidad de Agregar:

Lo que hacemos es casi idéntico a Buscar, recorremos los nodos y las listas, cuando en una lista no está la letra que buscamos la agregamos. Eso no modifica la complejidad porque recorrer toda la lista se come la complejidad de agregar un nodo. entonces también es $O(k_x|A|)$.

Complejidades del Trie

Vector

proc	complejidad
vacía	O(1)
tamaño	O(1)
agregar	O(k*|A|)
buscar	O(k)
eliminar	O(k)

Lista Enlazada

proc	complejidad
vacía	O(1)
tamaño	O(1)
agregar	O(k*|A|)
buscar	O(k*|A|)
eliminar	O(k*|A|)

La ventaja de una y otra está en la memoria que ocupan.

Para qué sirve un Trie?

Ejercicio 12. Dada una matriz binaria (de unos y ceros), identifique si existen filas repetidas. Debe realizar esto recorriendo la matriz sólo una vez.

Ejemplo:

```
Entrada:
[1  0  0  1  0]
[0  1  1  0  0]
[1  0  0  1  0]
[0  0  1  1  0]
[0  1  1  0  0]

Salida:
{2, 4} (la fila 2 es igual a la fila 0 y la fila 4 es igual a la fila 1)
```

Para qué sirve un Trie?

Cómo lo resolveriamos si lo intentabamos hace 4 horas?
recorremos cada fila y la comparamos con el resto:

```
para i desde 0 hasta n
    para j desde i+1 hasta n
        if (filasIguales?(i,j)){
            print( {i,j} )
```

Cual es la complejidad de filasIguales? O(n)
Cual es la complejidad de los dos “para”, sin tener en cuenta filasIguales?
n + (n-1) + (n-2) + … + 3 + 2 + 1 = n*(n-1)/2 = O(n²)
entonces estamos haciendo n² veces O(n) = O(n³)

Se puede mejorar?

Para qué sirve un Trie?

Agregamos cada fila como una clave a un Trie.
Cuando vemos que ya estaba definida es porque se repite.
Qué complejidad tiene?

Para qué sirve un Trie?

Agregamos cada fila como una clave a un Trie.
Cuando vemos que ya estaba definida es porque se repite.
Qué complejidad tiene?

Agregamos n filas así que $O(n)$.

Para qué sirve un Trie?

Ejercicio bonus:

Nos dan una matriz arbitrariamente grande.
Los valores de la matriz son aleatorios y están entre 0 y k, k un número dado.

Conviene arreglos o listas?

Y si los valores no están acotados?