Algoritmos y Estructuras de Datos

Algoritmos sobre palabras
Hashing

Tries

Algoritmos de búsqueda en texto

Hashing

Diccionarios

Un **diccionario** es un tipo de datos que sirve para asociar claves a valores, con las siguientes operaciones:

- Crear un diccionario vacío.
- **Insertar** una clave, asociándola a un valor.
- **Buscar** una clave, determinando el valor que tiene asociado.
- **Eliminar** una clave.

Diccionarios

Conocemos algunas estructuras para implementar diccionarios.
Si hay n claves y cada clave es un entero, las complejidades son:

	Inserción	**Búsqueda**	**Eliminación**
lista de pares	$O(n)$	$O(n)$	$O(n)$
arreglo ordenado	$O(n)$	$O(\log n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$

¿Qué pasaría si las claves son cadenas de texto de longitud m?

	Inserción	**Búsqueda**	**Eliminación**
lista de pares	$O(mn)$	$O(mn)$	$O(mn)$
arreglo ordenado	$O(mn)$	$O(m \log n)$	$O(mn)$
AVL	$O(m \log n)$	$O(m \log n)$	$O(m \log n)$

Veremos una estructura especializada para cadenas de texto.

Tries

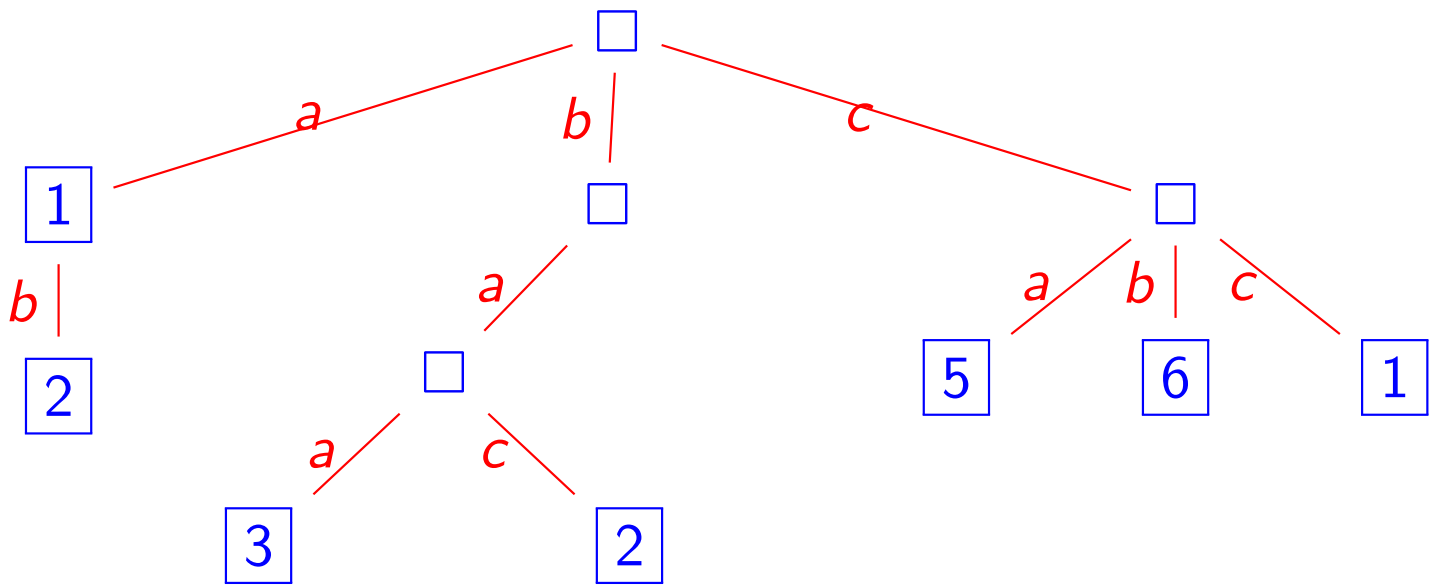
Sea Σ un *alfabeto* (conjunto finito de símbolos).

Un **trie** es un árbol que en cada nodo tiene $|\Sigma|$ hijos, uno por cada símbolo del alfabeto. Representa un diccionario.

1. El camino de la raíz hasta un nodo corresponde a una *clave*.
2. El valor almacenado en un nodo corresponde a un *valor*.

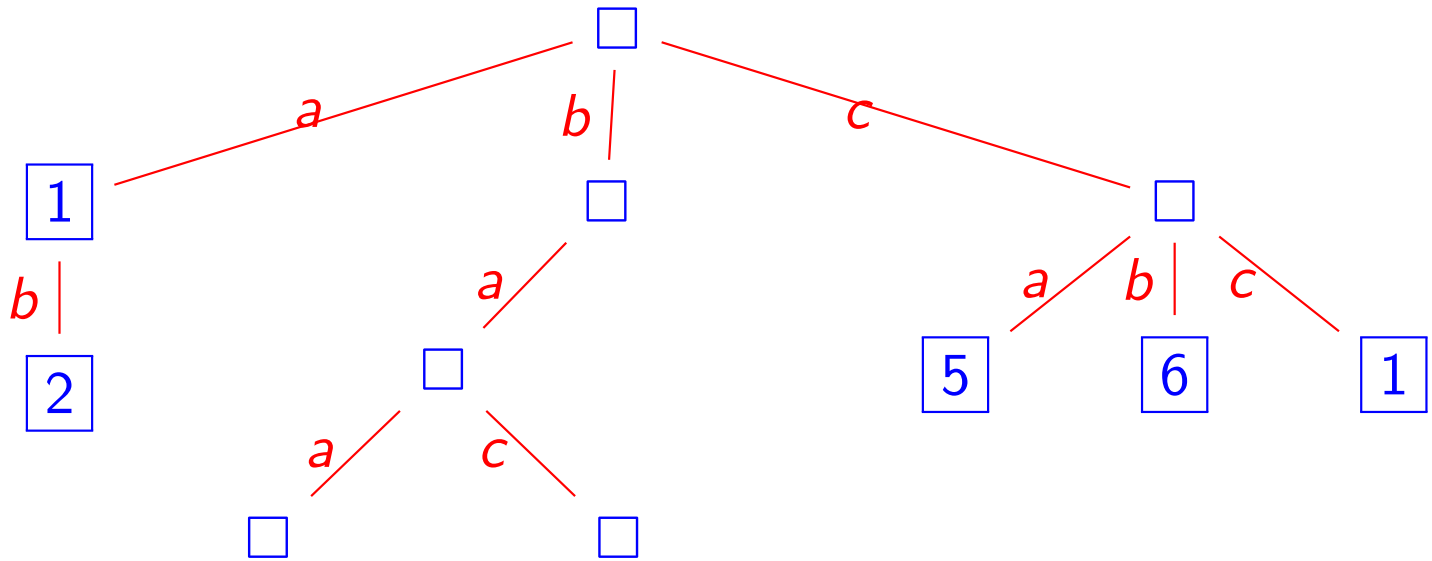
Ejemplo

Si $\Sigma = \{a, b, c\}$, el siguiente es un trie:



Tries

El siguiente también es un trie, que "desperdicia espacio":



Para evitar este problema, mantenemos el siguiente **invariante**:

- Las hojas (nodos sin hijos) deben tener un valor asociado.
- La única excepción es la raíz del trie.
  Permitimos que no tenga hijos ni un valor asociado.

Tries en Python

Necesitamos algunas operaciones sobre símbolos del alfabeto:

```
ALFABETO = "abcdefghijklmnopqrstuvwxyz"

def indice(simbolo):
    return ord(simbolo) - ord('a')
```

Representación de un trie en Python

Reservamos `None` para representar un trie vacío.

Cada nodo del trie es una instancia de la siguiente clase:

```
class Nodo:
    def __init__(self):
        self.hayClave = False
        self.valor = None
        self.hijos = [None for simbolo in ALFABETO]
```

Tries — Algoritmo de inserción

Para **insertar** una clave $a_0 \ldots a_{m-1}$ con un valor v en un trie:

- Ubicarse en la raíz del trie.
- Para cada i desde 0 hasta $m - 1$:
  - Bajar al hijo correspondiente al símbolo a_i.
    (En caso de que no exista, crearlo).
- Marcar el nodo para indicar que la clave está presente.
  Guardar el valor v en dicho nodo.

La complejidad temporal en peor caso es $O(m)$.

Observación interesante

La complejidad no depende del número total de claves.

Depende sólo de la longitud de la clave.

Tries — Algoritmo de inserción en Python

```
def insertar(trie, clave, valor):
    nodo = trie
    for simbolo in clave:
        i = indice(simbolo)
        if nodo.hijos[i] is None:
            nodo.hijos[i] = Nodo()
        nodo = nodo.hijos[i]
    nodo.hayClave = True
    nodo.valor = valor
```

Tries — Algoritmo de búsqueda

Para **buscar** una clave $a_0 \ldots a_{m-1}$ en un trie:

- Ubicarse en la raíz del trie.
- Para cada i desde 0 hasta $m - 1$:
  - Bajar al hijo correspondiente al símbolo a_i.
    (Si no existe, la clave no está presente).
- Mirar el nodo para determinar si la clave está presente.
  Si está presente, devolver también su valor asociado.

La complejidad temporal en peor caso es $O(m)$.

Tries — Algoritmo de búsqueda en Python

```
def buscar(trie, clave):
    nodo = trie
    for simbolo in clave:
        i = indice(simbolo)
        if nodo.hijos[i] is None:
            return (False, None)
        nodo = nodo.hijos[i]
    return (nodo.hayClave, nodo.valor)
```

Tries — Algoritmo de eliminación

Para **eliminar** una clave $a_0 \dots a_{m-1}$ de un trie:

- Ubicarse en la raíz del trie.
- Para cada i desde 0 hasta $m-1$:
  - Bajar al hijo correspondiente al símbolo a_i.
    (Si no existe, la clave no está presente).
- Marcar el nodo para indicar que su clave no está presente.
  Borrar el valor asociado en caso de que existiera.

Para preservar el invariante de que las hojas no están vacías.

- Armar una pila con todos los nodos recorridos.
  (Se modifica ligeramente el algoritmo de arriba).
- Mientras el nodo actual sea una hoja vacía y no sea la raíz:
  - Modificar el padre, borrándole el hijo que corresponda.
  - Ubicarse en dicho padre.

La complejidad temporal en peor caso es $O(m)$.

Tries — Algoritmo de eliminación en Python

```
def eliminar(trie, clave):
    pila = []
    nodo = trie
    for simbolo in clave:
        i = indice(simbolo)
        if nodo.hijos[i] is None:
            return
        pila.append(nodo)
        nodo = nodo.hijos[i]
    nodo.hayClave = False
    nodo.valor = None
    limpiarHojasVacias(pila, nodo, clave)
```

Tries — Algoritmo de eliminación en Python

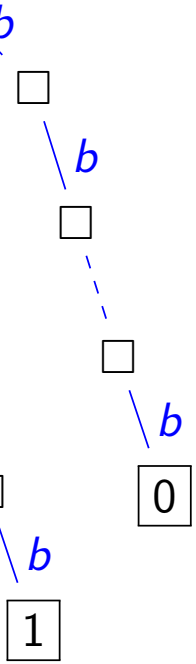
```
def limpiarHojasVacias(pila, nodo, clave):
    k = len(clave) - 1
    while len(pila) > 0 and esHojaVacia(nodo):
        padre = pila.pop()
        i = indice(clave[k])
        k -= 1
        padre.hijos[i] = None
        nodo = padre

def esHojaVacia(nodo):
    return nodo.valor is None and \
           all(hijo is None for hijo in nodo.hijos)
```

Tries — Complejidad espacial

Un trie con n palabras puede tener $O(n^2)$ nodos.

Por ejemplo, las palabras de la forma $a^i b^n$ con $0 \leq i \leq n$:



Tries — Comentario (árboles PATRICIA)

Se puede evitar la "explosión" en la cantidad de nodos comprimiendo caminos que no tienen bifurcaciones.

Tries

Algoritmos de búsqueda en texto

Hashing

Problema de *string matching*

Suponemos fijado un alfabeto Σ (conjunto finito de símbolos).
Notamos Σ^* al conjunto de las cadenas sobre Σ.

Descripción del problema

- **Entrada:** dos cadenas $p, q \in \Sigma^*$.
- **Salida:** un booleano que indica si q ocurre como subcadena de p. En el caso positivo, queremos identificar además el índice i de la primera ocurrencia de q en p.

Ejemplo

$$p = \underset{0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,10}{\text{“abracadabra”}} \qquad q = \text{“cad”} \qquad i = 4$$

Observación

- Si $|p| < |q|$, no puede haber ocurrencias de q en p.
- Típicamente $|p| >> |q|$.
  P. ej. hallar ocurrencias de “Rocinante” en Don Quijote.

Problema de *string matching*

Algoritmo ingenuo

Sean $p = a_0 \dots a_{n-1}$ y $q = b_0 \dots b_{m-1}$.
Queremos hallar la primera ocurrencia de q dentro de p.

- Para cada i desde 0 hasta $n - m$: $O(n - m)$ iteraciones.
  - Si $a[i \dots i + m) = q$: Comparar cuesta $O(m)$.
    - ✓ La primera ocurrencia de q en p es i.
- ✗ No hay ocurrencias de q en p.

Complejidad temporal en peor caso: $O(n\,m)$.

Problema de *string matching*

Observación

Cada comparación fallida implica trabajo desperdiciado.

Por ejemplo, si buscamos "`barbaros`" en "`barbarbaros`":

- La primera ocurrencia tentativa falla: "`barbar`baros".
- La segunda ocurrencia tiene éxito: "`barbarbaros`".
- Se pierde el trabajo de analizar las letras "`barbar`baros".

Motivación

- Nos gustaría reaprovechar el trabajo de los intentos que fallan.
- Nos gustaría avanzar en el texto sin tener que "retroceder".

Problema de *string matching*

Ejemplo

Si buscamos "`barbaros`" en un texto y vemos "`barbar...`":

- Si la siguiente letra es "o", el algoritmo sigue adelante.
- Si la siguiente letra no es "o", este intento falla.

  Pero las últimas tres letras que vimos ("`barbar...`") son un prefijo de la palabra que buscamos.

  Podríamos tratar de ver si la palabra sigue con "`...baros`".

Idea — sufijos que son prefijos

En general, si $q = b_0 \dots b_{m-1}$ es la palabra buscada, queremos:

- Para cada índice $0 \leq i < m$:
  ¿cuál es el **sufijo** más largo de $q[0 \dots i)$ que es **prefijo** de q?

Problema de *string matching*

Definición — tabla de falla

Sea $q = b_0 \dots b_{m-1}$ la palabra buscada.

La tabla de falla es un arreglo F tal que:

> $F[i]$ es el tamaño del sufijo propio más largo de $q[0 \dots i)$ que además es prefijo de q.

Más precisamente, para cada $0 \leq i \leq m$:

$$F[i] = \max\{k \mid k < i,\ q[i-k \dots i) = q[0 \dots k)\}$$

Tabla de falla

Ejemplo — tabla de falla

Si $q =$ “abracadabra”, donde $|q| = 11$:

0 1 2 3 4 5 6 7 8 9 10

i	$q[0..i)$	$F[i]$
0	“”	0
1	“a”	0
2	“ab”	0
3	“abr”	0
4	“abra”	1
5	“abrac”	0
6	“abraca”	1
7	“abracad”	0
8	“abracada”	1
9	“abracadab”	2
10	“abracadabr”	3
11	“abracadabra”	4

Algoritmo KMP

Supongamos ya calculada la tabla de falla F para $q = b_0 \dots b_{m-1}$.

Algoritmo KMP (Knuth–Morris–Pratt)

Sean $p = a_0 \dots a_{n-1}$ y $q = b_0 \dots b_{m-1}$.
Queremos hallar la primera ocurrencia de q dentro de p.

- Inicializar $j := 0$.
- Para cada i desde 0 hasta $n-1$: $O(n)$ iteraciones.
  - Mientras $j > 0$ y $p[i] \neq q[j]$: $O(n)$ iteraciones **en total**
    $j := F[j]$
  - Si $p[i] = q[j]$: $O(1)$
    $j := j + 1$.
  - Si $j = m$: $O(1)$
    ✓ La primera ocurrencia de q en p es $(i - m + 1)$.
- ✗ No hay ocurrencias de q en p.

Complejidad temporal en peor caso: $O(n)$.

Algoritmo KMP en Python

Asumimos ya calculada la tabla de falla:

```
def kmp(texto, cadena, tabla_falla):
    if len(cadena) == 0:
        return (True, 0)
    n = len(texto)
    m = len(cadena)
    j = 0
    for i in range(n):
        while j > 0 and texto[i] != cadena[j]:
            j = tabla_falla[j]
        if texto[i] == cadena[j]:
            j = j + 1
        if j == m:
            return (True, i + 1 - m)
    return (False, None)
```

Construcción de la tabla de falla

¿Cuál es el costo de construir la tabla de falla?

Asumimos que $|q| = m$.

- La construcción ingenua tiene complejidad $O(m^3)$:

```
def tabla_falla(q):
    m = len(q)
    f = [max(k for k in range(i) if q[i - k : i] == q[: k])
            for i in range(1, m + 1)]
    return [0] + f
```

  Puede ser aceptable cuando $n >> m$.

- Hay algoritmos para construir la tabla de falla en $O(m)$.

Tries

Algoritmos de búsqueda en texto

Hashing

Hashing

Las **tablas de hash** son una de las estructuras más usadas.
Sirven para implementar conjuntos / diccionarios.

Contras

- Las complejidades de inserción, búsqueda y eliminación son lineales $O(n)$ en peor caso.
- Pueden ser sensibles a la distribución de la entrada.

Pros

- Las complejidades de inserción, búsqueda y eliminación son constantes $O(1)$ en caso promedio.
  (Bajo hipótesis apropiadas de distribución de la entrada).
- Son muy sencillas de implementar.
- Se "llevan bien" con el hardware moderno.

Hashing

Motivación

Queremos implementar un diccionario. Es decir:

- dadas n claves $k_0, \ldots, k_{n-1} \in K$
- asociarlas a n valores.

Si tuviéramos una función $h : K \to \{0, 1, \ldots, n-1\}$ tal que:

- **no hay colisiones**, es decir,

  los índices $h(k_0), \ldots, h(k_{n-1})$ son distintos,

entonces alcanzaría con guardar un arreglo $[v_0, \ldots, v_{n-1}]$.

El valor de la clave k sería $v_{h(k)}$

Hashing

En la práctica raramente es posible evitar colisiones.

La idea es buscar una función $h : K \to \{0, \ldots, n-1\}$ que distribuya a las posibles claves de manera lo más uniforme posible.

La función h se llama **función de hash**.

Ejemplo

- Un sensor nos dice que en el tiempo t_i el valor registrado es y_i.
  El tiempo se mide en milisegundos desde el comienzo.
- Queremos registrar 1000 lecturas $(t_0, y_0), \ldots (t_{999}, y_{999})$.
  El período en el que se toman muestras es de 10 minutos.
- Función de hash "buena": $h(t) = (t \textbf{ mod } 1000)$.
- Función de hash "mala": $h(t) = (t \textbf{ div } 1000)$.

Hashing

¿Cómo lidiar con colisiones?

Hay varios métodos.

1. **Direccionamiento abierto:**
   Si el *bucket* de la posición $h(k)$ ya está ocupado:
   buscar linealmente un espacio a partir de la posición $h(k) + 1$.
2. **Direccionamiento cerrado:**
   Cada *bucket* contiene una lista de pares (clave, valor).
   Dentro de esa lista se hace búsqueda lineal.