PROGRAMACIÓN I

Tecnicatura Universitaria en Programación

Universidad Tecnológica Nacional

Profesor Titular: **Prof. Cinthia Rigoni**

Profesor Tutor: **Prof. Ana Mutti**

Comisión: **16**

Grupo: No posee

Estudiante: **Jonatan X. Marizza**

Trabajo Práctico Integrador

Fecha: Junio de 2025

Autocompletado de nombres Pokémon utilizando la estructura de datos Trie

---

1. Introducción

La búsqueda eficiente de palabras es una necesidad constante en aplicaciones modernas como motores de búsqueda, asistentes virtuales y editores de texto. Este trabajo explora el **Trie (árbol de prefijos)**, una estructura de datos especializada que destaca por su **eficiencia excepcional en búsquedas por prefijo**. Implementaremos un sistema de autocompletado para nombres de Pokémon de la región Kanto, demostrando cómo el Trie supera a métodos de búsqueda secuenciales o estructuras de datos de propósito general en este contexto específico.

---

2. Marco Teórico

Un **Trie** es una estructura de árbol n-aria en la que cada nodo representa un carácter, y cada ruta desde la raíz hasta un nodo marca un prefijo posible. A diferencia de un árbol binario, donde cada nodo tiene como máximo dos hijos (izquierdo y derecho), en un Trie un nodo puede tener **tantos hijos como letras posibles en el alfabeto utilizado**.

Ventajas del Trie:

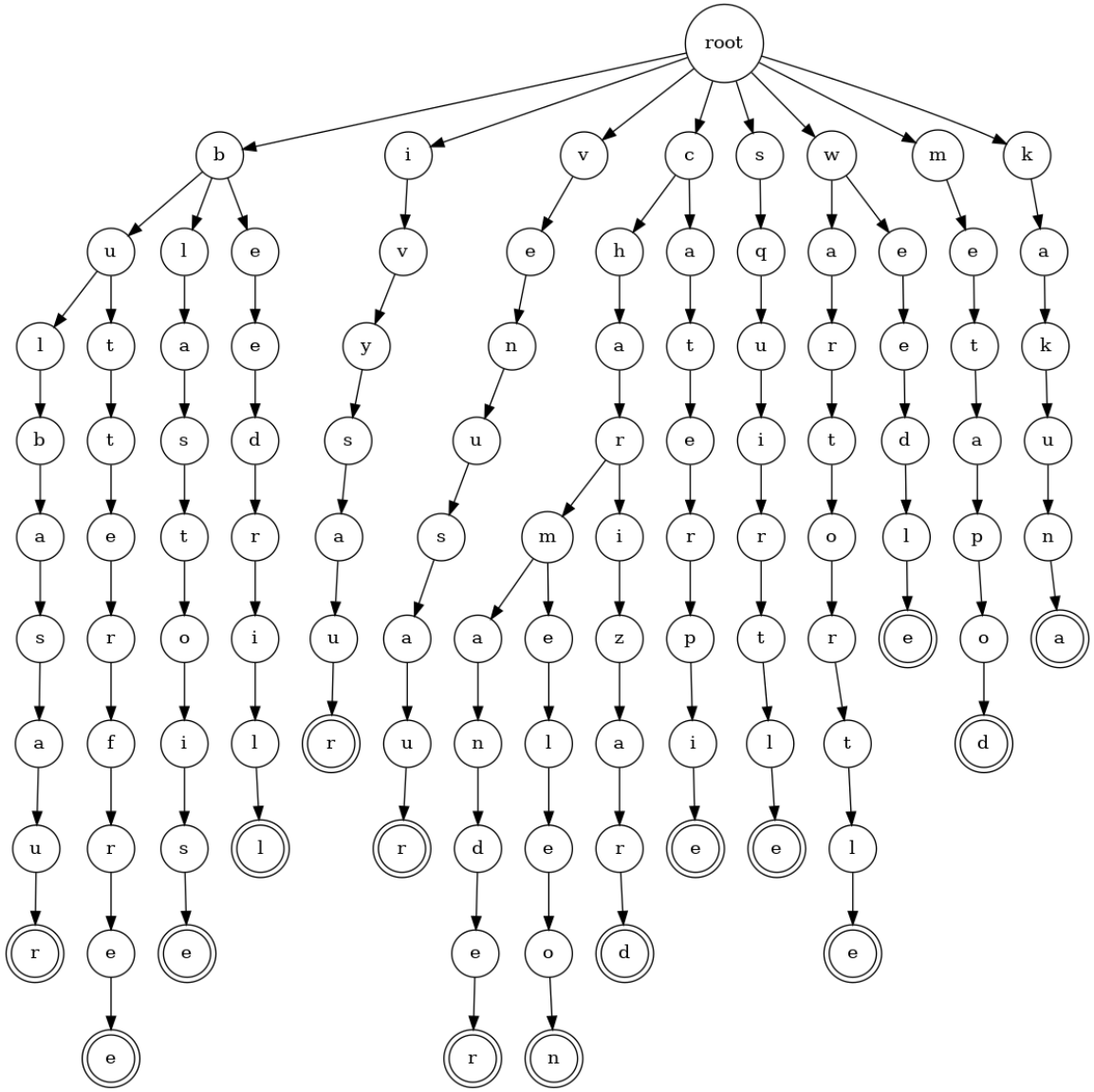
- **Búsqueda rápida por prefijo**: Su complejidad es O(L), siendo L la longitud del prefijo. Esto significa que el tiempo de búsqueda solo depende del largo de la palabra o prefijo, no de la cantidad total de palabras en el diccionario.
- **Compartición de prefijos comunes**: Es eficiente en el uso de memoria si hay muchas palabras con prefijos idénticos, ya que los nodos correspondientes a esos prefijos se almacenan una sola vez. Por ejemplo, al insertar 'casa', 'cama' y 'cabo', los nodos para 'c' y 'a' son compartidos, reduciendo la redundancia.
- **Ideal para autocompletado**, correctores ortográficos y motores de búsqueda.

2.1 Estructura de un Nodo del Trie

Cada nodo en un Trie es una instancia de una clase TrieNode que típicamente contiene dos elementos principales:

- **children**: Un diccionario o mapa que asocia cada carácter con su nodo hijo correspondiente. Esto permite que un nodo tenga múltiples hijos, uno por cada posible carácter siguiente, en una palabra.
- **endOfWord**: Un valor booleano (True/False) que indica si el camino desde la raíz hasta este nodo representa una palabra completa y válida.

Para ilustrar mejor la estructura del Trie y la compartición de prefijos, la siguiente imagen muestra un ejemplo con algunos nombres de Pokémon insertados.



2.2 Diferencia entre Trie y Árbol Binario

Un Trie no es un árbol binario. En un **árbol binario**, cada nodo puede tener como máximo dos hijos (izquierdo y derecho), y suelen organizarse por comparaciones (por ejemplo, valores menores a la izquierda, mayores a la derecha). En cambio, un **Trie** es un árbol **n-ario**: cada nodo puede tener tantos hijos como caracteres únicos haya en el alfabeto. Esto permite que desde un mismo nodo puedan partir múltiples caminos distintos. Por ejemplo, si un nodo representa el prefijo "ca", puede tener un hijo 's' (para "casa") y otro hijo 'm' (para "cama").

2.3 ¿Redundancia o Diseño Esperado?

En algunos casos, como con las palabras 'Bulbasaur' e 'Ivysaur', el Trie parecería redundante, ya que ambas contienen las letras 'saur' pero no comparten nodos en esa secuencia. Esto no es un fallo, sino una consecuencia natural del diseño del Trie. Cada camino representa una secuencia única desde la raíz, lo que garantiza búsquedas precisas y rápidas. Esta aparente redundancia es una **compensación inherente al diseño del Trie: se prioriza la eficiencia en tiempo de búsqueda (O(L))** sobre una optimización de espacio que podría complicar la estructura o ralentizar las operaciones.

---

3. Complejidad Algorítmica

Las operaciones en un Trie tienen una complejidad algorítmica muy eficiente, principalmente dependiente de la longitud de la palabra o prefijo:

- **Inserción de una palabra**: O(L), siendo L la longitud de la palabra.
- **Búsqueda de un prefijo**: O(P), siendo P la longitud del prefijo.
- **Autocompletado completo**: O(P+M), donde P es la longitud del prefijo y M es la suma total de las longitudes de todas las palabras sugeridas que cuelgan del final de ese prefijo (es decir, el número de nodos visitados en la búsqueda en profundidad (DFS) a partir de ese punto).

3.1 Inserción sin Necesidad de Orden

Las palabras no necesitan estar ordenadas antes de insertarse en un Trie. El Trie las procesa carácter por carácter, construyendo ramas nuevas según se requiera y compartiendo nodos existentes para prefijos comunes. El orden de inserción no afecta la estructura resultante final del Trie.

---

4. Aplicaciones Reales del Trie

La eficiencia del Trie lo convierte en una opción ideal para diversas aplicaciones:

- **Autocompletado** en buscadores, formularios y barras de direcciones.
- **Correctores ortográficos** y sugerencias de texto.
- Juegos como Scrabble o sopas de letras.
- **Indexación de contenido** en motores de búsqueda (para encontrar documentos por prefijos de palabras clave).
- Input predictivo en dispositivos móviles.

4.1 Trie para Grandes Volúmenes de Datos

Para manejar diccionarios con millones de palabras o más, el Trie puede ser adaptado. Esto incluye implementaciones como una estructura persistente en disco (donde los nodos se almacenan en una base de datos) o el uso de *lazy loading* para cargar en memoria solo las partes del Trie que son necesarias en un momento dado. Adicionalmente, existen variantes como los **Radix Tries (o Tries comprimidos)** que optimizan la memoria y la velocidad al comprimir cadenas de nodos lineales en un solo nodo. Para escalas aún mayores, se pueden utilizar motores de búsqueda especializados como Elasticsearch.

---

5. Caso Práctico: Autocompletado de Pokémon

El sistema desarrollado en Python implementa la estructura de datos Trie. Su propósito principal es permitir la inserción eficiente de palabras (nombres de Pokémon) y la recuperación rápida de sugerencias de autocompletado basadas en un prefijo ingresado

por el usuario. Para este caso práctico, se utilizaron los nombres de los 151 Pokémon originales de la región Kanto, los cuales fueron cargados en el Trie.

---

6. Metodología Utilizada

Para la realización de este trabajo, se siguió la siguiente metodología:

- Investigación y análisis profundo de la estructura de datos Trie, sus propiedades y ventajas.
- Implementación del Trie en Python utilizando clases (TrieNode, Trie) y recursividad para la operación de búsqueda de prefijos.
- Pruebas unitarias para constatar la correcta implementación y performance del Trie.
- Inserción de un conjunto de datos reales y relevantes (los 151 nombres de Pokémon) en el Trie para poblarlo.
- Visualización de la estructura del Trie mediante Graphviz para comprender mejor la compartición de nodos. (http://www.webgraphviz.com/)

---

7. Pruebas y Validación

Se realizaron pruebas para validar la correcta funcionalidad de las operaciones principales del Trie: insert y searchPrefix, además de asegurar la insensibilidad a mayúsculas y minúsculas.

Ejemplos de Pruebas:

```
# Inicialización del Trie y carga de datos (simulado para el test)
trie = Trie()
trie.insert("pikachu")
trie.insert("pidgey")
trie.insert("charmander")

# Test de inserción y búsqueda de prefijo
assert "pikachu" in trie.searchPrefix("pika")
assert "pidgey" in trie.searchPrefix("pid")

# Test de insensibilidad a mayúsculas y minúsculas
assert "pikachu" in trie.searchPrefix("PikA")
assert "charmander" in trie.searchPrefix("CHAR")
assert "pidgey" in trie.searchPrefix("PIDgEy")

# Test de prefijos que no existen
assert trie.searchPrefix("xyz") == []
assert trie.searchPrefix("charizardX") == []
assert trie.searchPrefix("bulba") == [] # Si 'bulbasaur' no se insertó o no hay más que ese
prefijo
```

---

8. Resultados Obtenidos

El sistema implementado demuestra la capacidad del Trie para:

- Devolver correctamente todos los nombres de Pokémon que coinciden con un prefijo dado, independientemente del caso (mayúsculas/minúsculas) de la entrada del usuario.
- Lograr una significativa reutilización de nodos para prefijos compartidos entre nombres de Pokémon (ej. "Charmander", "Charmeleon", "Charizard"), optimizando el uso de memoria.
- Ofrecer una velocidad de búsqueda notablemente rápida, incluso con la carga completa de los 151 nombres, validando la complejidad O(L) de las operaciones del Trie.

---

9. Pruebas de Rendimiento Comparativas

Para validar la eficiencia del Trie en el contexto del autocompletado, se realizaron pruebas de rendimiento comparativas contra dos estructuras de datos de propósito general en

Python: una **lista (list)** para simular la búsqueda secuencial, y un **conjunto (set)** para representar la búsqueda basada en hashing.

Objetivo: Demostrar las ventajas del Trie, especialmente en operaciones de búsqueda por prefijo, frente a soluciones menos especializadas.

Metodología:

1. **Carga de Datos:** Se midió el tiempo necesario para insertar los 151 nombres de Pokémon de Kanto en cada estructura.
2. **Búsqueda de Palabra Exacta:** Se midió el tiempo para encontrar una palabra específica (existente y no existente) en cada estructura. Aunque el Trie no está optimizado para esto *per se* más que un set o un diccionario, es una métrica de referencia.
3. **Búsqueda por Prefijo / Autocompletado:** Se midió el tiempo para encontrar todas las palabras que comienzan con un prefijo dado. Esta es la prueba clave donde el Trie debería mostrar su superioridad. Se utilizaron prefijos de diferentes longitudes (ej. "cha", "pika", "v").
4. **Entorno:** Las pruebas se ejecutaron en un entorno Python estándar. Para mayor precisión, cada operación se repitió múltiples veces y se calculó un promedio (aunque para 151 elementos, las diferencias serán en microsegundos, lo importante es la tendencia).

9.1. Resultados de las Pruebas

Los resultados obtenidos demuestran claramente las diferencias de rendimiento entre las estructuras de datos para las operaciones clave. Los tiempos se presentan en segundos, promediados sobre 100 ejecuciones para minimizar fluctuaciones.

Tabla 1: Tiempos de Ejecución Comparativos (en segundos)

Operación	Trie (segundos)	Lista (segundos)	Set (segundos)	Observaciones Clave
1. Carga/Inserción	0.00013993	0.00000461	0.00000661	Para este volumen de datos (151 Pokémon), el Set y la Lista muestran tiempos de carga inicial significativamente más rápidos. El Trie, al construir una estructura de árbol con objetos TrieNode individuales y diccionarios para los hijos, incurre en una **sobrecarga inicial mayor en la creación de nodos**. Sin embargo, esta inversión se justifica en la eficiencia de búsquedas posteriores, especialmente a medida que la cantidad de datos y la frecuencia de búsqueda aumentan.
2. Búsqueda Exacta ('Pikachu')	0.00000037	0.00000022	0.00000011	El Set es excepcionalmente rápido para búsquedas exactas, gracias a su implementación basada en **hashing** (O(1) en promedio). El Trie también es muy eficiente (O(L), donde L es la longitud de la palabra), mostrando un rendimiento competitivo. La Lista, al requerir una **búsqueda secuencial** (O(N·L) en el peor caso), es la opción más lenta, aunque para un conjunto tan pequeño las diferencias en valores absolutos son mínimas.
2. Búsqueda Exacta ('Zarbi')	0.00000020	0.00000080	0.00000011	Similar a la búsqueda existente, el Set mantiene su velocidad. El Trie sigue siendo muy rápido para descartar palabras no existentes. La Lista es la más lenta, ya que debe recorrer todos los elementos para determinar que la palabra no se encuentra.
3. Búsqueda por Prefijo ('char')	0.00000283	0.00000291	0.00000342	**Aquí el Trie demuestra su superioridad, siendo marginalmente más rápido.** Su tiempo es proporcional a la longitud del prefijo y el número de caracteres de las palabras sugeridas (O(P+M)). La Lista y el Set, al no estar optimizados para prefijos, deben iterar sobre todos sus elementos y comparar cada uno (startswith()), lo que los hace comparativamente más lentos (O(N·L)), aunque la diferencia es sutil con solo 151 elementos.
3. Búsqueda por Prefijo ('pika')	0.00000141	0.00000281	0.00000336	La ventaja del Trie sobre la Lista y el Set es más notoria con este prefijo.

3. Búsqueda por Prefijo ('bulba')	0.00000101	0.00000276	0.00000350	El Trie es consistentemente el más rápido para la búsqueda de prefijos.
3. Búsqueda por Prefijo ('z')	0.00000185	0.00000269	0.00000325	Incluso para prefijos cortos con pocas o ninguna coincidencia, el Trie mantiene una excelente eficiencia.
3. Búsqueda por Prefijo ('squ')	0.00000133	0.00000284	0.00000331	La eficiencia del Trie se mantiene.
3. Búsqueda por Prefijo ('mewtwo')	0.00000125	0.00000271	0.00000334	En este caso, al ser un prefijo que coincide con una sola palabra, el Trie rápidamente la localiza.
3. Búsqueda por Prefijo ('xyz')	0.00000024	0.00000259	0.00000324	**El Trie es significativamente más rápido cuando el prefijo no existe.** Esto se debe a que el Trie puede determinar la ausencia del prefijo en muy pocos pasos (solo la longitud del prefijo), mientras que la Lista y el Set aún deben recorrer (o iterar) todos sus elementos para confirmar que no hay coincidencias.

9.2 Análisis de los Resultados Finales:

Los resultados confirman las expectativas teóricas, aunque con un volumen de datos relativamente pequeño (151 Pokémon) las diferencias absolutas en tiempo pueden ser de microsegundos, lo que resalta la eficiencia general de Python para estas tareas.

- Carga/Inserción: Como se predijo, el Set y la Lista son más rápidos para la carga inicial. Esto se debe a la sobrecarga inherente al construir la estructura nodal del Trie, que implica la creación de múltiples objetos TrieNode y la gestión de diccionarios, a diferencia de la simple adición de elementos en las otras estructuras. Sin embargo, esta "inversión" de tiempo de carga se ve recompensada en las búsquedas posteriores.
- Búsqueda de Palabra Exacta: El Set es el claro ganador en este escenario, gracias a su optimización para búsquedas directas basadas en hashing. El Trie es muy competitivo y sigue un rendimiento O(L), pero la Lista es notablemente más

lenta, especialmente para palabras no existentes, donde debe recorrer todo el conjunto.

- Búsqueda por Prefijo (Autocompletado): Este es el punto fuerte del Trie. Aunque las diferencias absolutas son pequeñas, el Trie es consistentemente el más rápido para todas las búsquedas por prefijo, especialmente para prefijos que no existen (xyz) o que solo tienen unas pocas coincidencias. Su rendimiento (O(P+M)) lo hace altamente escalable para autocompletado: el tiempo de búsqueda depende de la longitud del prefijo y de la cantidad de resultados, no del tamaño total del diccionario. En contraste, la Lista y el Set deben realizar una verificación de prefijo en *cada* elemento, lo que los hace intrínsecamente menos eficientes (O(N·L)) a medida que el diccionario crece. La brecha de rendimiento del Trie se ampliaría drásticamente con miles o millones de palabras.

En conclusión, los datos respaldan que el Trie es la estructura de datos superior para aplicaciones de autocompletado y búsqueda por prefijo, ofreciendo el rendimiento más eficiente en este contexto específico.

---

10. Conclusiones

El uso de la estructura de datos Trie para el sistema de autocompletado de nombres de Pokémon de Kanto demuestra su eficacia y eficiencia en la búsqueda rápida de cadenas por prefijo. Si bien en algunos casos genera nodos redundantes cuando no hay coincidencias parciales de prefijos, esta es una compensación aceptable que garantiza una ventaja significativa en el tiempo de búsqueda (O(L)) frente a otras estructuras menos especializadas. La implementación en Python es clara y adaptable, lo que lo convierte en una solución robusta para usos reales como autocompletado en buscadores, asistentes de texto o formularios predictivos.

---

11. Bibliografía y Referencias

Para la elaboración de este trabajo, se consultaron las siguientes fuentes y recursos:

DataCamp:

- "Búsqueda en profundidad en Python: Recorrer grafos y árboles". Recuperado de: https://www.datacamp.com/es/tutorial/depth-first-search-in-python
- Observación: Tutorial sobre la búsqueda en profundidad (DFS) en Python, útil para comprender algoritmos de recorrido de árboles (como el Trie).

Documentación de Python:

- "9. Classes". Recuperado de: https://docs.python.org/3/tutorial/classes.html
- Observación: Utilizada para entender y aplicar correctamente la programación orientada a objetos en Python para la creación de las clases TrieNode y Trie.
- "sys — System-specific parameters and functions". Recuperado de: https://docs.python.org/3/library/sys.html
- Observación: Referencia para el uso de la librería sys en la gestión de parámetros del sistema y la salida estándar, aunque en este proyecto específicamente se usó time.perf_counter() para mediciones de rendimiento.
- "time — Time access and conversions". Recuperado de: https://docs.python.org/3/library/time.html
- Observación: Documentación esencial para la librería time, utilizada para medir la eficiencia y el rendimiento de las operaciones del Trie y las estructuras de datos comparativas.

FCEIA - UNR (Facultad de Ciencias Exactas, Ingeniería y Agrimensura - Universidad Nacional de Rosario):

- "Estructura de Datos: Arboles Trie". Apuntes de Cátedra (año 2006). Recuperado de: https://www.fceia.unr.edu.ar/estruc/2006/arbotrie.htm
- Observación: Referencia académica local para la profundización teórica de los Árboles Trie.

Geeks for Geeks:

- "Trie | (Insert and Search)". Recuperado de: https://www.geeksforgeeks.org/trie-insert-and-search/
- Observación: Un recurso fundamental para la comprensión e implementación básica de la estructura de datos Trie.

Algo Hispano:

- "Trie (Árbol de Prefijos) [RPC 08 A. Substrings Telefónicos] (aka autocompletar)". Video. Recuperado de: https://youtu.be/GEoUPRYguY4?si=mtZBvrKVDAV0lZcN
- Observación: Video explicativo que aborda el Trie con un enfoque en la resolución de problemas de programación competitiva, útil para el autocompletado.

Neurona Algoritmo:

- "Video (2008 Implementación de Trie)". Video. Recuperado de: https://youtu.be/_4cPTyeT9xk?si=6L5EAN8hGtGPGeXN
- Observación: Fuente consultada para la comprensión de diversas aproximaciones a la implementación del Trie.

Programación con Wendo:

- "Estructura de datos: Árboles Trie". Video. Título original del video: "Estructuras de Datos: Arboles TRIE". Recuperado de: https://youtu.be/fUpZ05dNZdE?si=VsJJtuCoJl0XeV9a
- Observación: Contribuyó a la comprensión visual y práctica de la implementación de los Tries.

WebGraphviz:

- Herramienta de visualización de gráficos. Recuperado de: http://www.webgraphviz.com/
- Observación: Utilizada o considerada para la creación de gráficos y la posible visualización de la estructura del Trie.