

Trabajos Integradores - Programación I

Datos Generales

Investigación Aplicada: Implementación de un Diccionario mediante Árboles Binarios de Búsqueda en Python

- Comisión : 12
- Integrantes :
- Matias Costantini e Ivan Daniliuk
- Profesor: Ariel Enferrel
- Tutor Luciano Chiroli
- Fecha de entrega: 7-6-25

1. Introducción

Las estructuras de datos son pilares fundamentales en la programación, esenciales para organizar y gestionar información de forma eficiente. Entre ellas, los árboles destacan por su capacidad para modelar relaciones jerárquicas, lo que los hace ideales para resolver problemas de búsqueda y almacenamiento de datos. En esta investigación, nos enfocamos en los Árboles Binarios de Búsqueda (ABB).

La elección de este tema para nuestra investigación surge de la necesidad fundamental en la informática de gestionar grandes volúmenes de datos de forma eficiente. Un ejemplo claro es la construcción de un diccionario digital. Los diccionarios requieren la capacidad de almacenar palabras y sus definiciones, buscarlas rápidamente y presentarlas de manera ordenada. Los ABB ofrecen una solución elegante y comprensible para este problema, permitiendo una aplicación directa de sus principios. Su estructura facilita tanto la recuperación ágil de información como la iteración ordenada de sus elementos, lo cual es ideal para un diccionario.

Los ABB son fundamentales para comprender cómo se construyen y operan estructuras de datos ordenadas, sirviendo de base para algoritmos de búsqueda con complejidad logarítmica ($O(\log n)$ en el caso promedio), lo que representa una mejora sustancial frente a la búsqueda lineal en grandes conjuntos de datos.

Con el desarrollo de este trabajo, nos proponemos alcanzar los siguientes objetivos: implementar un diccionario funcional en Python utilizando un ABB; comprender y aplicar las operaciones fundamentales de inserción y búsqueda en esta estructura; analizar la eficiencia de dichas operaciones en el contexto del diccionario; y demostrar la capacidad de listar las palabras almacenadas en estricto orden alfabético

2. Marco Teórico

¿Qué es un árbol?

Un árbol es una estructura de datos no lineal que organiza la información de manera jerárquica. A diferencia de las estructuras lineales como las listas enlazadas, que siguen una secuencia lógica, los árboles presentan una organización ramificada, donde cada nodo puede apuntar a uno o varios nodos descendientes. Si bien comparten con las listas enlazadas la característica de poseer "punteros" o referencias que conectan elementos, la disposición de los datos en un árbol no es secuencial, sino que emula la forma de un árbol genealógico o un organigrama, con un elemento principal del que se derivan los demás [Def. cátedra apunte teórico].

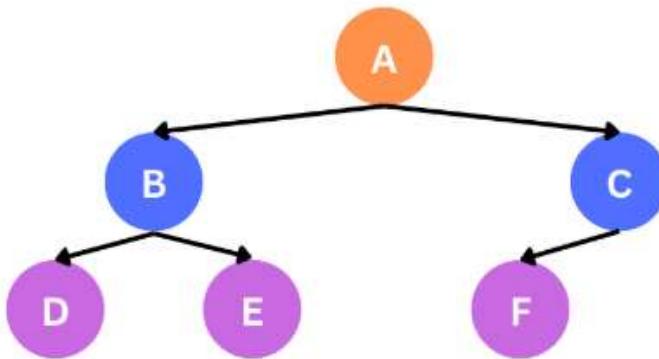


Figura 1. Representación de grafos de un árbol. Los círculos representan nodos y las flechas relaciones entre dichos nodos.

Terminología Básica de Árboles

- **Nodo Raíz:** Nodo principal del que derivan todos los demás.
- **Nodo Padre:** Nodo que tiene descendientes.
- **Nodo Hijo:** Nodo que desciende de otro.
- **Nodos Hermanos:** Nodos que comparten el mismo parentesco.
- **Hojas (Nodos Terminales):** Nodos sin hijos.
- **Nodos Internos (No Terminales):** Nodos que tienen al menos un hijo.
- **Arista Rama:** Conexión entre dos nodos.
- **Camino:** Secuencia de nodos conectados por aristas.

- **Longitud del Camino:** Número de aristas en un camino.

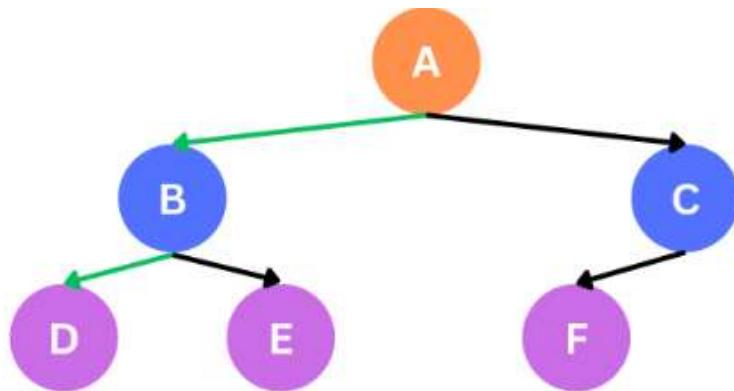


Figura 2. La longitud de camino entre los nodos A y D es igual a 2 ya que para llegar desde A hasta D hay que atravesar las dos ramas resaltadas en verde.

- **Nivel de un Nodo:** Longitud del camino desde la raíz hasta el nodo. El nivel de la raíz es 0.
 - **Altura de un Nodo:** Longitud del camino más largo desde el nodo hasta una hoja.
 - **Altura del Árbol:** Altura del nodo raíz.
 - **Profundidad de un Nodo:** Longitud del camino desde la raíz hasta el nodo (similar al nivel).
 - **Subárbol:** Un árbol formado por un nodo y todos sus descendientes.
 - **Grado de un Nodo:** Número de hijos que tiene un nodo.
 - **Grado del Árbol:** El máximo grado de los nodos del árbol.

Tipos Comunes de Árboles

Árboles Binarios

Un **árbol binario** es una estructura de datos jerárquica donde cada nodo tiene como máximo dos hijos: un hijo izquierdo y un hijo derecho. No es obligatorio que un nodo tenga ambos hijos; puede tener solo uno o ninguno. La conexión entre un nodo padre y sus hijos es direccional, lo que significa que solo se puede navegar hacia abajo en la jerarquía del árbol.

El nodo superior se conoce como raíz y es el punto de entrada al árbol. Los nodos que no tienen hijos se llaman nodos hoja (o terminales).

Árboles Binarios de Búsqueda (ABB / BST - Binary Search Tree)

Un Árbol Binario de Búsqueda (ABB), comúnmente conocido por sus siglas en inglés **BST** (**Binary Search Tree**), es una especialización de los árboles binarios que impone una propiedad de ordenación fundamental:

Para cada nodo, todas las claves (o palabras, en nuestro caso) en su subárbol izquierdo son menores que la clave del propio nodo, y todas las claves en su subárbol derecho son mayores. Esta propiedad de ordenación permite que las operaciones de búsqueda, inserción y eliminación sean muy eficientes.

Los ABB son muy relevantes para almacenar datos ordenados y facilitar búsquedas rápidas.

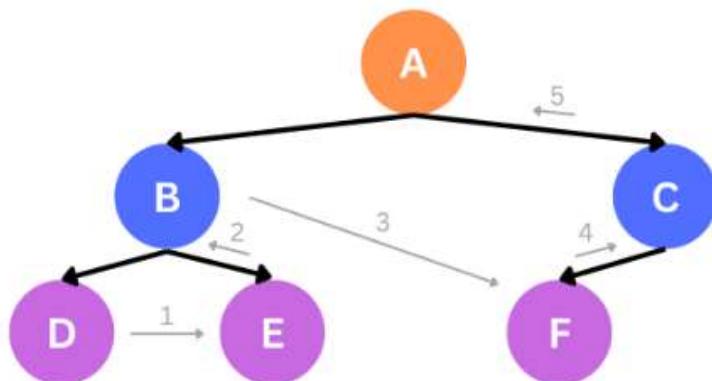


Figura 7. Para recorrer el árbol de la imagen en postorden partimos del nodo hoja que está más a la izquierda (D), visitamos su nodo hermano (E) y subimos hacia el padre de ambos (B). Una vez que terminamos de recorrer el subárbol izquierdo, comenzamos a recorrer el subárbol derecho, partiendo desde su nodo hoja ubicado más a la izquierda (F) y luego su padre (C). Finalmente visitamos el nodo raíz (A). El orden resultante es DEBFCA.

Árboles AVL

Un Árbol **AVL** es un Árbol Binario de Búsqueda auto-balanceable que garantiza una eficiencia de $O(\log n)$ para todas las operaciones (búsqueda, inserción y eliminación), incluso en el peor de los casos.

Los árboles **AVL** mantienen su equilibrio asegurando que, para cada nodo, la diferencia de altura entre su subárbol izquierdo y su subárbol derecho (su factor de equilibrio) nunca es mayor que 1. Si esta propiedad se viola después de una inserción o eliminación, el árbol realiza rotaciones (simples o dobles) para restaurar el equilibrio.

Árboles B y B+

Cuando se trata de gestionar grandes volúmenes de datos almacenados en disco, como los que se encuentran en bases de datos, las estructuras de árbol tradicionales (**como los árboles binarios de búsqueda**) se vuelven inefficientes. Esto se debe a que el acceso al disco es significativamente más lento que el acceso a la memoria RAM. Cada "salto" para leer un nodo individual podría implicar una costosa operación de E/S (Entrada/Salida) al disco.

Para resolver este problema, se desarrollaron árboles especializados, siendo los **B-Trees** y sus variantes como los **B+Trees** las soluciones predominantes.

- Los **B-Trees** son árboles de búsqueda auto-balanceados diseñados para maximizar la cantidad de información leída en una sola operación de disco, permitiendo que cada nodo contenga múltiples claves y tenga múltiples hijos. Esto reduce drásticamente la altura del árbol, minimizando los accesos a disco.
- Los **B+Trees** son una evolución de los B-Trees, optimizados para bases de datos. En un B+Tree, todos los datos se almacenan exclusivamente en los nodos hoja, los cuales están enlazados entre sí para permitir un recorrido secuencial eficiente. Los nodos internos sólo contienen claves para guiar la búsqueda.

Árboles Trie (Árbol de Prefijos).

Un **Trie** es una estructura de datos de árbol que se utiliza para almacenar un diccionario de cadenas. La principal característica que lo distingue es que los nodos del árbol representan prefijos comunes de las cadenas almacenadas. Cada nodo, excepto la raíz, almacena un único carácter, y el camino desde la raíz hasta un nodo particular forma una cadena o prefijo. Los **Tries** son especialmente eficientes para búsquedas de prefijos y funcionalidades de autocompletado.

Operaciones Fundamentales en Árboles (con foco en ABB)

Las operaciones fundamentales en cualquier tipo de árbol permiten manipular y extraer información de la estructura. Para los **Árboles Binarios de Búsqueda (ABB)**, estas operaciones son especialmente importantes por su eficiencia en el manejo de datos ordenados.

Recorridos en un ABB

- **Preorden (Raíz, Izquierda, Derecha):** Útil para copiar el árbol.
- **Inorden (Izquierda, Raíz, Derecha):** Fundamental para los ABB, ya que permite recuperar las palabras del Diccionario en orden alfabético.
- **Postorden (Izquierda, Derecha, Raíz):** Útil para eliminar el árbol.

- **Recorrido por Nivel (Amplitud):** Menos común para funcionalidades de diccionario, pero es un recorrido estándar

Implementación de Árboles en Python

La implementación de árboles en Python generalmente implica definir una clase `Nodo` y luego una clase que encapsule la lógica del árbol (por ejemplo, `ArbolBinarioBusqueda` o `DiccionarioArbol`).

- Representación de Nodos: Una clase `Nodo` suele tener atributos para la `clave` (ej., la palabra), la `definición` (opcional), y referencias a su `hijo_izquierdo` y `hijo_derecho`.
- Estructura del Árbol: La clase `ArbolBinarioBusqueda` encapsularía la lógica para las operaciones como la inserción, búsqueda y eliminación de nodos.

3. Caso Práctico: Diccionario con Árbol Binario de Búsqueda

En esta sección se detalla la implementación de un sistema de diccionario utilizando un Árbol Binario de Búsqueda (ABB) en Python. El sistema permitirá agregar nuevas palabras con sus definiciones, buscar palabras existentes y mostrar todas las palabras del diccionario en orden alfabético.

3. Descripción del Problema

El objetivo es desarrollar un programa que simule un diccionario digital. Este diccionario debe permitir:

1. **Agregar nuevas palabras:** Cada palabra se almacenará junto con su definición.
2. **Buscar palabras:** El usuario podrá ingresar una palabra y el sistema deberá mostrar su definición si existe en el diccionario.
3. **Listar Todas las palabras:** Mostrar todas las palabras almacenadas en el diccionario en orden alfabético.
4. **(Opcional) Modificar definición:** Permitir cambiar la definición de una palabra existente.
5. **(Opcional) Eliminar palabras:** Permitir quitar palabras del diccionario.

La eficiencia en la búsqueda y la capacidad de listar palabras ordenadamente son requisitos clave.

Diseño de la Solución con Árboles Binarios de Búsqueda

Se ha elegido un **Árbol Binario de Búsqueda (ABB)** para implementar el diccionario debido a sus propiedades:

- **Ordenamiento inherente:** Las palabras se almacenarán de forma que un recorrido *inorden* del árbol las devolverá en orden alfabético. Esto satisface el requisito de listar palabras ordenadamente.
- **Búsqueda eficiente:** En un ABB balanceado (o en promedio), la búsqueda de una palabra tiene una complejidad de $O(\log n)$, donde n es el número de palabras en el diccionario. Esto es significativamente más rápido que una búsqueda lineal ($O(n)$) en una lista no ordenada.
- **Inserción eficiente:** Similar a la búsqueda, la inserción también tiene una complejidad promedio de $O(\log n)$.

Cada **nodo** del ABB almacenará:

- palabra (string): La palabra en sí (actuará como clave).
- definición (string): La definición de la palabra.
- hijo izquierdo (referencia a Nodo): Subárbol con palabras alfabéticamente menores.
- hijo derecho (referencia a Nodo): Subárbol con palabras alfabéticamente mayores.

La estructura del diccionario estará representada por una clase Diccionario ABB que contendrá una referencia al nodo raíz del árbol y los métodos para las operaciones requeridas.

Explicación de Decisiones de Diseño

- **Elección del ABB:** Se optó por un ABB por su equilibrio entre simplicidad de implementación (comparado con árboles auto-balanceables como AVL o Rojo-Negro) y su buen rendimiento promedio para las operaciones requeridas en un diccionario (búsqueda, inserción $O(\log n)$ y listado ordenado $O(n)$ mediante recorrido *inorden*).
- **Manejo de Mayúsculas/Minúsculas:** Las palabras se convierten a minúsculas antes de almacenarlas y al buscarlas. Esto asegura que la búsqueda sea insensible a mayúsculas y minúsculas (ej: "Árbol" y "árbol" se consideran la misma palabra). La visualización puede capitalizar la palabra.
- **Palabras Duplicadas:** En la implementación actual, si se intenta agregar una palabra que ya existe, se podría optar por:
 1. No hacer nada.
 2. Actualizar la definición existente. (Esta sería una buena opción para un diccionario real).

3. Permitir duplicados (generalmente no deseado para claves de diccionario). La decisión a tomar (ej: actualizar definición) debe justificarse.

- **Recursión:** Se utilizan métodos recursivos para las operaciones de inserción, búsqueda y recorrido inorden, ya que reflejan de manera natural la estructura recursiva de los árboles y resultan en un código más claro y conciso para estas tareas.
- **Alternativas (Trie):** Se consideró brevemente el uso de un árbol Trie, que es altamente eficiente para diccionarios y búsquedas de prefijos. Sin embargo, para los objetivos de este trabajo práctico centrado en la comprensión de los árboles binarios de búsqueda, el ABB se consideró más apropiado y directo. Un Trie podría ser una excelente extensión futura.

3.5. Validación del Funcionamiento

Para validar el correcto funcionamiento del diccionario implementado mediante un Árbol Binario de Búsqueda (ABB), se realizarán las siguientes pruebas, ejecutando el script diccionario.py e interactuando con su menú:

1. Prueba de Agregar Palabra y Definición:

- **Descripción:** Verificar que se puedan agregar nuevas palabras y sus definiciones correctamente al diccionario.
- **Pasos a Realizar:**
 1. Ejecutar el script diccionario.py.
 2. Seleccionar la opción "1. Agregar nueva palabra".
 3. Ingresar la palabra: Manzana
 4. Ingresar la definición: Fruta pomácea comestible, de forma redonda y sabor más o menos dulce, según la variedad.
 5. Verificar el mensaje de confirmación: "'Manzana' agregada al diccionario."
- **Resultado Esperado:** La palabra "Manzana" y su definición se almacenan en el diccionario. No hay salida directa en este paso más que el mensaje de confirmación.

```
○ $ python diccionario.py
--- Diccionario Interactivo ---
1. Agregar nueva palabra
2. Buscar palabra
3. Mostrar todas las palabras (orden alfabético)
4. Salir
Seleccione una opción: 1
Ingrese la palabra: Manzana
Ingrese la definición: Fruta pomácea comestible, de forma redonda y sabor más o menos dulce, según la variedad.
'Manzana' agregada al diccionario.
```

2. Prueba de Buscar Palabra Existente (Caso Exacto):

- **Descripción:** Comprobar que se puede buscar y encontrar una palabra previamente agregada, utilizando el mismo caso de escritura con el que (visualmente) se agregó.
- **Pasos a Realizar (continuación de la prueba 1):**
 1. Seleccionar la opción "2. Buscar palabra".
 2. Ingresar la palabra a buscar: Manzana
- **Resultado Esperado:** El programa debe mostrar: Definición: Manzana: Fruta pomácea comestible, de forma redonda y sabor más o menos dulce, según la variedad.

```
--- Diccionario Interactivo ---
1. Agregar nueva palabra
2. Buscar palabra
3. Mostrar todas las palabras (orden alfabético)
4. Salir
Seleccione una opción: 2
Ingrese la palabra a buscar: Manzana
Definición: Fruta pomácea comestible, de forma redonda y sabor más o menos dulce, según la variedad.
```

3. Prueba de Buscar Palabra Existente (Caso Diferente):

- **Descripción:** Verificar que la búsqueda no sea sensible a mayúsculas y minúsculas, encontrando una palabra independientemente del caso utilizado en la búsqueda.
- **Pasos a Realizar (continuación de la prueba 1):**
 1. Seleccionar la opción "2. Buscar palabra".
 2. Ingresar la palabra a buscar: manzana (todo en minúsculas).
- **Resultado Esperado:** El programa debe mostrar: Definición: Manzana: Fruta pomácea comestible, de forma redonda y sabor más o menos dulce, según la variedad.
 1. Seleccionar la opción "2. Buscar palabra".
 2. Ingresar la palabra a buscar: MANZANA (todo en mayúsculas).
- **Resultado Esperado:** El programa debe mostrar: Definición: Manzana: Fruta pomácea comestible, de forma redonda y sabor más o menos dulce, según la variedad.

```
--- Diccionario Interactivo ---
1. Agregar nueva palabra
2. Buscar palabra
3. Mostrar todas las palabras (orden alfabético)
4. Salir
Seleccione una opción: 2
Ingrese la palabra a buscar: manzana
Definición: Fruta pomácea comestible, de forma redonda y sabor más o menos dulce, según la variedad.
```

4. Prueba de Buscar Palabra Inexistente:

- **Descripción:** Asegurar que el programa maneje adecuadamente la búsqueda de palabras que no están en el diccionario.
- **Pasos a Realizar:**
 1. Ejecutar el script diccionario.py (o continuar si ya hay palabras).
 2. Seleccionar la opción "2. Buscar palabra".

3. Ingresar la palabra a buscar: Pera (asumiendo que "Pera" no ha sido agregada).
- **Resultado Esperado:** El programa debe mostrar: Definición: 'Pera' no se encontró en el diccionario.

```
--- Diccionario Interactivo ---
1. Agregar nueva palabra
2. Buscar palabra
3. Mostrar todas las palabras (orden alfabético)
4. Salir
Seleccione una opción: 2
Ingrese la palabra a buscar: Pera
Definición: 'Pera' no se encontró en el diccionario.
```

5. Prueba de Agregar Múltiples Palabras y Listar en Orden Alfabético:

- **Descripción:** Verificar que se puedan agregar múltiples palabras y que la funcionalidad de listar todas las palabras las muestre en orden alfabético.
- **Pasos a Realizar:**
 1. Ejecutar el script diccionario.py.
 2. Agregar las siguientes palabras (usando la opción 1):
 - Palabra: Banana, Definición: Fruta amarilla alargada.
 - Palabra: Árbol, Definición: Planta perenne, de tronco leñoso y elevado.
 - Palabra: Casa, Definición: Edificación construida para ser habitada.
 - Palabra: Zebra, Definición: Animal équido africano con rayas.
 3. Seleccionar la opción "3. Mostrar todas las palabras (orden alfabético)".

Resultado Esperado: El programa debe mostrar las palabras y definiciones en el siguiente orden:

--- Palabras en el Diccionario ---

Arbol: Planta perenne, de tronco leñoso y elevado.

Banana: Fruta amarilla alargada.

Casa: Edificación construida para ser habitada.

Zebra: Animal équido africano con rayas.

```
---- Diccionario Interactivo ----
1. Agregar nueva palabra
2. Buscar palabra
3. Mostrar todas las palabras (orden alfabético)
4. Salir
Seleccione una opción: 3

---- Palabras en el Diccionario ---
Arbol: Planta perenne, de tronco leñoso y elevado.
Banana: Fruta amarilla alargada.
Casa: Edificación construida para ser habitada.
Zebra: Animal équido africano con rayas.
```

6. Prueba de Listar Palabras en un Diccionario Vacío:

- **Descripción:** Comprobar el comportamiento de la opción de listar palabras cuando el diccionario está vacío.
- **Pasos a Realizar:**
 1. Ejecutar el script diccionario.py (asegurándose que esté vacío o reiniciándolo).
 2. Seleccionar la opción "3. Mostrar todas las palabras (orden alfabético)".

Resultado Esperado: El programa debe mostrar:

```
--- Palabras en el Diccionario ---
```

El diccionario está vacío.

```
---- Diccionario Interactivo ----
1. Agregar nueva palabra
2. Buscar palabra
3. Mostrar todas las palabras (orden alfabético)
4. Salir
Seleccione una opción: 3

---- Palabras en el Diccionario ---
El diccionario está vacío.
```

7. Prueba de Salida del Programa:

- **Descripción:** Verificar que la opción de salir termine la ejecución del programa.
- **Pasos a Realizar:**
 1. Ejecutar el script diccionario.py.

2. Seleccionar la opción "4. Salir".
- **Resultado Esperado:** El programa debe mostrar el mensaje Saliendo del diccionario. ¡Hasta luego! y terminar su ejecución.

```
--- Diccionario Interactivo ---
1. Agregar nueva palabra
2. Buscar palabra
3. Mostrar todas las palabras (orden alfabético)
4. Salir
Seleccione una opción: 4
Saliendo del diccionario. ¡Hasta luego!
idani@LAPTOP-T2L9GEJA MINGW64 ~/OneDrive - ITBA/UTN/03 Primer Cuatrimestre - 1 año/04 Programación 1/00 Trabajo final integrador/Proyecto integrador
$
```

8. Prueba de Opción Inválida en el Menú:

- **Descripción:** Asegurar que el menú maneje entradas no válidas de forma adecuada.
- **Pasos a Realizar:**
 1. Ejecutar el script diccionario.py.
 2. Ingresar una opción que no esté en el menú, por ejemplo: 5.
- **Resultado Esperado:** El programa debe mostrar el mensaje Opción no válida. Intente de nuevo. y volver a mostrar el menú.

```
--- Diccionario Interactivo ---
1. Agregar nueva palabra
2. Buscar palabra
3. Mostrar todas las palabras (orden alfabético)
4. Salir
Seleccione una opción: 5
Opción no válida. Intente de nuevo.
```

4. Metodología Utilizada

Para la realización de esta investigación aplicada sobre la implementación de un diccionario mediante Árboles Binarios de Búsqueda (ABB) en Python, se siguió una metodología estructurada que abarcó desde la fundamentación teórica hasta la validación práctica de la solución desarrollada. Esta metodología se puede desglosar en las siguientes fases:

4.1. Revisión Bibliográfica y Fundamentación Teórica

Inicialmente, se realizó una revisión de los conceptos fundamentales relacionados con las estructuras de datos, con un enfoque particular en los árboles. Se investigó la definición de árboles, su terminología básica (nodo raíz, padre, hijo, hojas, etc.), y se exploraron diversos tipos de árboles como los Árboles Binarios, Árboles Binarios de Búsqueda (ABB), Árboles AVL, Árboles B y B+, y Árboles Trie. Se prestaron especial atención a las operaciones fundamentales en ABB, como los recorridos (preorden, inorder, postorden), y a las consideraciones para su implementación en Python. Esta fase fue crucial para comprender la idoneidad de los ABB para el problema de construir un diccionario digital.

4.2. Definición del Problema y Alcance

Con base en la comprensión teórica, se definió claramente el problema a resolver: el desarrollo de un programa que simule un diccionario digital. Los requisitos funcionales establecidos fueron:

- Permitir agregar nuevas palabras junto con sus definiciones.
- Permitir la búsqueda de palabras para obtener su definición.
- Permitir listar todas las palabras almacenadas en orden alfabético. Se consideraron como opcionales la modificación de definiciones y la eliminación de palabras, aunque la implementación final se centró en las tres funcionalidades principales.

4.3. Diseño de la Solución

Para la implementación del diccionario, se eligió un **Árbol Binario de Búsqueda (ABB)** como estructura de datos principal. Esta decisión se basó en el ordenamiento inherente que facilita un recorrido inorden para listar palabras alfabéticamente y la eficiencia en las operaciones de búsqueda e inserción (complejidad promedio de $O(\log n)$).

El diseño contempló:

- **Clase Nodo:** Para representar cada entrada del diccionario, almacenando la palabra (como clave), su definición, y referencias a los hijos izquierdo y derecho.
- **Clase DiccionarioABB:** Para encapsular la lógica del árbol, incluyendo la referencia al nodo raíz y los métodos para las operaciones del diccionario.

Se tomaron decisiones de diseño específicas, como:

- **Manejo de mayúsculas/minúsculas:** Las palabras se convierten a minúsculas antes de ser almacenadas y durante la búsqueda para asegurar la insensibilidad al caso. La visualización, sin embargo, capitaliza la palabra.
- **Palabras duplicadas:** La implementación actual no realiza ninguna acción si se intenta agregar una palabra que ya existe (es decir, no actualiza la definición ni permite duplicados estrictos).
- **Uso de recursión:** Se emplearon métodos recursivos para las operaciones de inserción, búsqueda y recorrido, debido a su afinidad con la estructura jerárquica de los árboles.

Se consideró el Árbol Trie como alternativa, pero se descartó para este trabajo práctico para mantener el enfoque en la comprensión de los ABB.

4.4. Implementación en Python

La solución diseñada se implementó en el lenguaje de programación Python, utilizando el paradigma de Programación Orientada a Objetos. Se crearon las clases Nodo y DiccionarioABB conforme al diseño.

- La clase Nodo incluye un constructor que almacena la palabra en minúsculas y un método `__str__` para su representación legible, capitalizando la palabra.

- La clase DiccionarioABB implementa los métodos agregar_palabra, buscar_palabra y listar_palabras_inorden, utilizando métodos auxiliares recursivos (_agregar_recursivo, _buscar_recursivo, _inorden_recursivo) para las operaciones sobre el árbol.
- Se desarrolló una interfaz de usuario interactiva basada en consola mediante las funciones mostrar_menu() y main(), permitiendo al usuario seleccionar las operaciones a realizar.

El código fuente completo de esta implementación se encuentra en el archivo diccionario.py.

4.5. Pruebas y Validación

Para asegurar el correcto funcionamiento del diccionario, se diseñó y ejecutó un plan de pruebas detallado. Este plan, descrito en la sección "3.5. Validación del Funcionamiento" del presente documento, incluyó:

- Prueba de agregar una palabra y su definición.
- Pruebas de buscar palabras existentes, tanto con el caso exacto como con casos diferentes (mayúsculas/minúsculas).
- Prueba de buscar una palabra inexistente.
- Prueba de agregar múltiples palabras y listarlas en orden alfabético.
- Prueba de listar palabras en un diccionario vacío.
- Prueba de la opción de salida del programa.
- Prueba del manejo de opciones inválidas en el menú.

Las pruebas se realizaron ejecutando el script diccionario.py e interactuando directamente con el menú de opciones, verificando que los resultados obtenidos coincidieran con los resultados esperados.

4.6. Herramientas Utilizadas

- **Lenguaje de Programación:** Python 3.x.
- **Entorno de Desarrollo:** Un editor de texto estándar o Entorno de Desarrollo Integrado (IDE) compatible con Python.
- **Sistema Operativo:** Cualquier sistema operativo con un intérprete de Python funcional (ej. Windows, Linux, macOS).
- **Consola/Terminal:** Para la ejecución del script y la interacción con el programa.

5. Resultados Obtenidos

La realización de este trabajo práctico permitió alcanzar los objetivos propuestos, logrando la implementación exitosa de un diccionario digital funcional utilizando Árboles Binarios de Búsqueda (ABB) en Python. A continuación, se detallan los aspectos que funcionaron correctamente y las dificultades o limitaciones presentadas.

Aspectos Funcionales Correctos

El programa diccionario.py demostró un funcionamiento correcto en sus funcionalidades principales, conforme a los requisitos establecidos:

- **Agregación de Palabras y Definiciones:** Se pueden agregar nuevas palabras con sus respectivas definiciones al diccionario. El sistema almacena las palabras en minúsculas para asegurar la consistencia interna, tal como se diseñó.
- **Búsqueda de Palabras:** La búsqueda de palabras es funcional y no sensible a mayúsculas/minúsculas, devolviendo la definición correspondiente si la palabra existe. Si la palabra no se encuentra, el programa informa adecuadamente al usuario.
- **Listado Alfabético de Palabras:** El programa puede listar todas las palabras y sus definiciones almacenadas en el diccionario en estricto orden alfabético. Esto se logra mediante un recorrido inorden del ABB, mostrando la palabra capitalizada para una mejor presentación visual.
- **Manejo del Diccionario Vacío:** Las operaciones de búsqueda y listado manejan correctamente el caso en que el diccionario no contenga ninguna palabra, informando de esta situación al usuario.
- **Interfaz de Usuario por Consola:** El menú interactivo resultó ser una forma efectiva de guiar al usuario a través de las diferentes funcionalidades del diccionario. El sistema gestiona adecuadamente las opciones válidas e informa sobre entradas incorrectas.
- **Salida del Programa:** La opción para salir del programa funciona como se esperaba, finalizando la ejecución de la aplicación.

En general, la implementación práctica sirvió para consolidar la comprensión de la estructura de datos del Árbol Binario de Búsqueda y sus operaciones fundamentales (inserción, búsqueda y recorrido).

Dificultades y Limitaciones Presentadas

Si bien el programa cumple con los objetivos básicos, durante su desarrollo y análisis se identificaron ciertas dificultades inherentes al alcance del proyecto y algunas limitaciones en la implementación actual:

- **Manejo de Palabras Duplicadas:** La presente implementación no actualiza la definición de una palabra si se intenta agregar nuevamente; simplemente no se realiza una nueva inserción. Esta es una limitación reconocida, y una mejora futura podría ofrecer al usuario la opción de actualizar la entrada existente.
- **Ausencia de Funcionalidades Avanzadas:** El diccionario carece de funcionalidades como la eliminación de palabras o la modificación de definiciones existentes. Estas características fueron consideradas fuera del alcance principal de este trabajo práctico enfocado en la comprensión de los ABB.
- **Persistencia de Datos:** Toda la información (palabras y definiciones) se almacena en memoria y, por lo tanto, se pierde al cerrar el programa. No se implementó un mecanismo para guardar o cargar el diccionario desde un archivo.
- **Eficiencia con Grandes Volúmenes y Datos Ordenados:** Aunque los ABB ofrecen buen rendimiento promedio, su eficiencia puede degradarse a O(n) en el peor de los casos (por ejemplo, al insertar datos ya ordenados), llevando a un árbol desbalanceado. Para aplicaciones a gran escala, se podría considerar el uso de

árboles auto-balanceables como los AVL, aunque esto añadiría complejidad a la implementación.

- **Interfaz de Usuario Básica:** La interfaz de usuario es exclusivamente a través de la consola, lo cual es funcional pero menos amigable en comparación con una interfaz gráfica de usuario (GUI).
- **Manejo de Errores:** Aunque se manejan opciones de menú inválidas, el manejo de otros tipos de errores de entrada (por ejemplo, definiciones vacías) es limitado. Podría mejorarse la validación de las entradas del usuario.

Enlace al repositorio Git:

<https://github.com/MatiasCostantini094/TPI1>

6. Conclusiones

Este trabajo práctico ha permitido la aplicación de conceptos teóricos sobre Árboles Binarios de Búsqueda (ABB) en un caso práctico y funcional: la implementación de un diccionario digital en Python. A lo largo del proyecto, se han alcanzado los objetivos propuestos y se han superado diversos desafíos inherentes al desarrollo de software y al manejo de estructuras de datos.

Dificultades que surgieron durante el proyecto y cómo se resolvieron

Durante el desarrollo del diccionario, se presentaron algunos desafíos y decisiones de diseño que fueron abordadas de la siguiente manera:

- **Manejo de Palabras Duplicadas:** Se planteó la cuestión de cómo actuar al intentar agregar una palabra que ya existía en el diccionario. La dificultad radicaba en decidir si se debía actualizar la definición, ignorar la nueva entrada o señalar un error.
 - **Resolución:** Para mantener la simplicidad y centrarse en las operaciones básicas del ABB, se optó por no modificar la entrada existente si la palabra ya se encontraba en el diccionario. El código de inserción (`_agregar_recursivo`) simplemente no realiza ninguna acción si la palabra ya es igual a la del nodo actual.
- **Complejidad de Funcionalidades Avanzadas:** La implementación de operaciones como la eliminación de nodos de un ABB (que requiere considerar múltiples casos según el número de hijos del nodo) o la modificación de entradas existentes representaba una complejidad adicional.
 - **Resolución:** Se decidió delimitar el alcance del proyecto, excluyendo estas funcionalidades avanzadas para concentrar los esfuerzos en la correcta implementación y comprensión de la inserción, búsqueda y recorrido inorden, que son fundamentales en los ABB.
- **Persistencia de Datos:** Se consideró la necesidad de que los datos del diccionario persistieran entre sesiones de uso, lo cual implicaría la implementación de lectura y escritura en archivos.
 - **Resolución:** Esta funcionalidad se identificó como una mejora deseable pero fuera del alcance principal del trabajo práctico, que se centró en la manipulación de la estructura de datos en memoria.

- **Potencial Desbalanceo del Árbol:** Una dificultad teórica de los ABB simples es su susceptibilidad al desbalanceo si los datos se insertan en un orden específico (por ejemplo, ya ordenados), lo que podría degradar el rendimiento de las operaciones a $O(n)$.
 - **Resolución:** Se reconoció esta limitación y se optó por utilizar un ABB simple por su menor complejidad de implementación, adecuada para los objetivos de aprendizaje del proyecto. Se mencionó la existencia de árboles auto-balanceables (como AVL) como una solución más robusta para escenarios con grandes volúmenes de datos o patrones de inserción adversos.
- **Diseño de la Interfaz de Usuario:** Se buscó una forma sencilla para que el usuario interactuara con el diccionario.
 - **Resolución:** Se implementó una interfaz de usuario basada en un menú de consola, que, aunque básica, resultó suficiente para probar y demostrar las funcionalidades del diccionario de manera efectiva.
- **Manejo de Errores de Entrada:** Asegurar que el programa maneje todas las posibles entradas incorrectas del usuario.
 - **Resolución:** Se implementó un manejo básico de errores para opciones de menú no válidas. Se reconoció que un manejo de errores más exhaustivo (por ejemplo, para entradas vacías en palabras o definiciones) podría ser una mejora futura, pero se priorizó la lógica del ABB.
- **Consistencia en el Manejo de Mayúsculas/Minúsculas:** Era importante definir cómo se tratarían las palabras ingresadas con diferentes combinaciones de mayúsculas y minúsculas.
 - **Resolución:** Se adoptó una estrategia clara: almacenar todas las palabras en minúsculas y convertir también a minúsculas los términos de búsqueda. Esto asegura que la búsqueda sea insensible al caso, mientras que para la visualización, las palabras se presentan capitalizadas para una mejor legibilidad.
- **Comprendión y Aplicación de Recursión:** El uso efectivo de la recursión es clave en las operaciones con árboles.
 - **Resolución:** A través del estudio y la práctica, se implementaron con éxito las funciones recursivas para la inserción, búsqueda y el recorrido inorden, como se evidencia en el código diccionario.py.

Cumplimiento de los objetivos planteados en la introducción

Los objetivos definidos al inicio de este trabajo práctico se han cumplido satisfactoriamente:

- **Implementar un diccionario funcional en Python utilizando un ABB:** Se desarrolló un programa en Python (diccionario.py) que implementa un diccionario utilizando un Árbol Binario de Búsqueda como estructura de datos subyacente, cumpliendo con el principal objetivo técnico.
- **Comprender y aplicar operaciones de inserción y búsqueda:** Las funciones agregar_palabra y buscar_palabra implementan la lógica de inserción y búsqueda en un ABB. Su correcto funcionamiento fue verificado mediante pruebas, demostrando la aplicación práctica de estos conceptos.

- **Analizar la eficiencia de estas operaciones:** Aunque no se realizaron pruebas de rendimiento empíricas exhaustivas, el marco teórico y las secciones de diseño del documento analizan la eficiencia teórica de las operaciones en un ABB ($O(\log n)$ en promedio, $O(n)$ en el peor caso). Se comprendió la implicancia del balanceo del árbol en el rendimiento.
- **Listar todas las palabras en orden alfabético:** La funcionalidad de listar palabras se implementó mediante un recorrido inorden del ABB (listar_palabras_inorden), lo que garantiza que las palabras se muestren siempre en orden alfabético, tal como se requería.

En conclusión, el proyecto no solo resultó en una aplicación funcional, sino que también sirvió como una valiosa herramienta de aprendizaje para afianzar los conocimientos sobre una de las estructuras de datos más importantes en ciencias de la computación. Las dificultades encontradas fueron gestionadas mediante decisiones de diseño conscientes y la delimitación del alcance, permitiendo alcanzar los objetivos pedagógicos y técnicos propuestos.

7. Bibliografía

- https://docs.google.com/document/d/10k16oL15EeyOaq92aoi4qwK3t_22X29-FSV2iV-8N1U/edit?tab=t.0 (Material propuesto por la catedra)
- https://www.youtube.com/playlist?list=PLy5wpwhsM-2IiY-qefALJ4K_XAhLZ2I- (Videos propuestos por la cátedra)
- https://www.youtube.com/watch?v=UG5dL-YpD1s&list=PLOw7b-NX043aiYaKBMd3k1IUZr8cXTzE&index=79&ab_channel=CharlyCimino

8. Anexos

Enlace al video - También incluido en el README

<https://youtu.be/9xbzKVqvzbE>