

Trabajo Integrador - Programación

Tema: Estructuras de datos avanzadas - árboles

Alumnos

Berrone Lanza Lina Lucia

Bayurk Mara Valentina

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación I

Docente Titular:

Prof. Cinthia Rigoni

Docente Tutor:

Brian Lara

08 de Junio de 2025

Índice	
1.Introducción	3
2. Marco teórico	4
1.Definición de árboles	4
i. Tipos de árboles, formas de búsqueda	5
ii. Árboles binarios en código: listas anidadas y diccionarios	6
iii. Propiedades de los árboles	7
iv. Aplicaciones de árboles	9
2. Forma de recorrer los árboles	10
a. Recursividad e iteración	12
3. Caso Práctico	13
1. Implementación con listas	14
2. Implementación con diccionarios	18
3. Casos extras	22
4. Metodología utilizada	26
5. Resultados Obtenidos	27
6. Conclusiones	28
7. Bibliografía	29
8. Link al video	30

1. Introducción

Elegimos desarrollar este trabajo integrador sobre estructuras de datos avanzados, específicamente los árboles, porque su estructura refleja muchos escenarios reales, como árboles genealógicos, organigramas o sistemas de archivos. Si bien al principio puede parecer complejo, consideramos que comprender cómo funcionan los árboles nos permite pensar de manera más estructurada y eficiente al momento de resolver problemas, especialmente cuando se trata de manejar grandes volúmenes de datos o de organizar la información.

Los árboles son muy útiles cuando necesitamos organizar información que tiene una estructura jerárquica, como una relación de padre e hijo. Un ejemplo claro de esto son los menús de navegación en una página web, las carpetas dentro de una computadora o los árboles de decisión que se usan en inteligencia artificial. Este tipo de estructura ayuda a ver con más claridad cómo están conectados los datos entre sí, y también a recorrerlos de forma ordenada. Por ejemplo, en una computadora, dentro de una carpeta puede haber otras carpetas o archivos, y eso se puede representar como un árbol. También pasa en los árboles de decisión, donde cada paso depende del anterior y nos lleva por diferentes caminos. Nos pareció interesante trabajar con esta estructura justamente porque está presente en muchas cosas que usamos todos los días.

Con este trabajo nos proponemos alcanzar tres objetivos principales:

- Comprender el funcionamiento básico de los árboles binarios y otras variantes.
- Aplicar este conocimiento en un caso práctico utilizando el lenguaje Python.
- Demostrar cómo los árboles pueden ayudarnos a organizar y procesar datos de manera más eficiente.

2. Marco Teórico

2.1 DEFINICIÓN DE LOS ÁRBOLES

Los árboles son una de las estructuras de datos más fundamentales y versátiles en informática. Su utilidad se extiende a múltiples áreas, ya que permiten representar relaciones jerárquicas de manera eficiente. A diferencia de otras estructuras como listas o diccionarios, que son lineales o basadas en clave-valor, los árboles están diseñados para situaciones en las que los elementos mantienen una organización en niveles o ramas.

En el mundo real, los árboles tienen aplicaciones muy variadas como modelar estructuras como los sistemas de archivos de una computadora, los árboles genealógicos, las bases de datos jerárquicas, etc. Esta estructura ramificada permite que un nodo principal (llamado raíz) se conecte con varios nodos secundarios (hijos), que a su vez pueden tener sus propios descendientes, generando así una estructura de múltiples niveles.

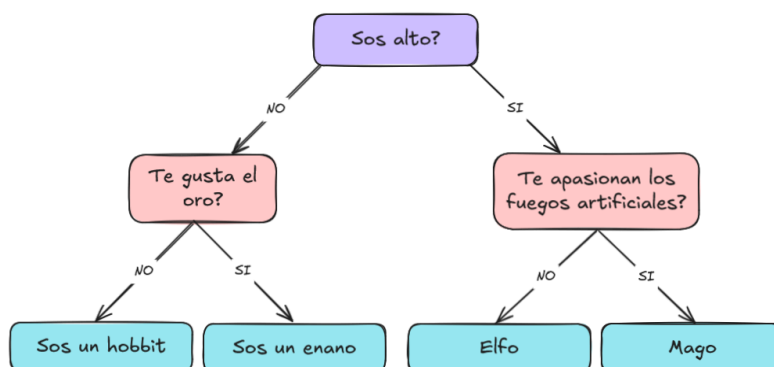
Asimismo, los árboles son un paso intermedio hacia estructuras más complejas como los grafos, donde las conexiones entre los nodos no necesariamente siguen una jerarquía, sino que pueden ser arbitrarias, como ocurre en redes sociales, rutas de navegación o redes de comunicación.

En programación, comprender cómo funcionan los árboles es esencial para desarrollar algoritmos más eficientes, tanto en tiempo de ejecución como en uso de memoria. Por ejemplo, en bases de datos, los árboles de búsqueda binaria permiten realizar búsquedas, inserciones y eliminaciones de forma rápida.

En resumen, el estudio de los árboles no solo permite resolver problemas específicos con jerarquías claras, sino que también brinda una base conceptual sólida para abordar estructuras de datos más avanzadas. Su implementación y aplicación son esenciales para cualquier persona que busque mejorar la eficiencia y organización de sus programas.

ELEMENTOS DE LOS ÁRBOLES

Los árboles se representan mediante grafos, siendo los nodos su unidad básica que se conectan entre sí mediante ramas; los nodos tienen dos componentes básicos: su VALOR y un puntero o nodos hijos.



Dependiendo de dónde está ubicado un nodo, puede recibir nombres especiales. Existen dos formas de clasificar nodos: según su ubicación en el árbol y según su relación con otros nodos.

En el dibujo superior, donde mostramos el árbol de decisiones que luego vamos a presentar en código y ahora pasamos a clasificar sus nodos:

Según su ubicación en el árbol

- **Nodo raíz:** es el punto de entrada a la estructura, generalmente ubicado en la parte más superior del árbol. No tiene un nodo padre y es único, es decir, cada árbol tiene un solo nodo raíz. En el ejemplo el nodo raíz es de color violeta y su valor es "SOS ALTO?"
- **Nodo rama o interno:** es cualquier nodo que tiene un padre y, al menos, un hijo. A diferencia de un nodo hoja, un nodo rama no es terminal y sigue propagando la jerarquía del árbol hacia abajo. En el gráfico los nodos internos son los naranjas con las preguntas "te gusta el oro?" y "te apasionan los fuegos artificiales"
- **Nodo hoja:** es cualquier nodo que no tenga hijos, es decir, un nodo terminal del árbol. Los nodos hoja están en los extremos de la estructura y no tienen más ramificaciones debajo de ellos. En el gráfico son los nodos azules con los valores de raza. ("Hobbit", "Elfo", "Enano" y "Mago")

Según su relación con otros nodos

Nodo Padre: es aquél que tiene uno o más nodos hijos conectados a él. Cada nodo, excepto la raíz, tiene solamente un nodo padre.

- El nodo **violeta** es padre de los nodos **naranja**
- El nodo **naranja_izquierdo** es padre de los nodos **azules (enano y hobbit)**.
- El nodo **naranja_derecho** es padre de los nodos **azules (mago y elfo)**.

Nodo hijo: un nodo hijo es aquél que está conectado con un nodo padre de modo que se encuentra directamente debajo de él en la jerarquía. El nodo hijo depende del nodo padre y no puede existir sin él. Puede, a su vez, convertirse en padre de otros nodos.

- Los nodos **ENANO Y HOBBIT** son hijos del nodo **TE GUSTA EL ORO?**.
- Los nodos **ELFO Y MAGO** son hijos del nodo **TE APASIONAN LOS FUEGOS ARTIFICIALES?**.
- Los nodos **NARANJAS** son hijos del nodo **SOS ALTO?**.

Nodo hermano: los nodos hermanos son aquellos que comparten el mismo nodo padre. Es decir, tienen la misma distancia respecto al nodo raíz y están en el mismo nivel en la jerarquía del árbol.

- Los nodos **NARANJAS** son hermanos entre sí.
- Los nodos **AZULES** son hermanos entre sí.

2.1.i Tipos de árboles

A continuación veremos los diferentes tipos de árboles que fueron encontrados en la investigación:

- **Árbol Binario:** Cada nodo puede tener como máximo dos hijos, comúnmente llamados hijo izquierdo e hijo derecho. Es la base de muchos algoritmos por su estructura simple y eficiente. Este es el que vamos a desarrollar
- **Árbol Binario de Búsqueda (BST):** Es un tipo especial de árbol binario donde para cada nodo, todos los valores en su subárbol izquierdo son menores al nodo, y todos los valores en su subárbol derecho son mayores. Esto permite una búsqueda, inserción y eliminación eficiente en tiempo logarítmico promedio.

- **Árbol N-ario:** Generaliza el árbol binario permitiendo que cada nodo tenga hasta N hijos. Es útil para representar estructuras con múltiples opciones o ramas, como menús o árboles de decisión complejos.
- **Árbol AVL:** Es un árbol binario de búsqueda auto-balanceado. La diferencia de altura entre los subárboles izquierdo y derecho de cualquier nodo (también llamado factor de equilibrio) nunca supera 1. Esto garantiza tiempos de operación consistentes y eficientes.
- **Árbol Trie (o árbol de prefijos):** Es un árbol especializado para la búsqueda de palabras. Cada nodo representa una letra, y el recorrido desde la raíz hasta un nodo terminal forma una palabra. Se usa comúnmente en correctores ortográficos, sistemas de autocompletado, y diccionarios digitales.

2.1.ii Árboles binarios en código: Listas anidadas y diccionarios

Los árboles pueden representarse en código mediante diferentes estructuras, siendo las más comunes las **listas anidadas** y los **diccionarios**:

- **Listas Anidadas:** Se representan como una lista donde el primer elemento es la información del nodo y el segundo elemento es una lista de sus hijos. Es adecuada para árboles de decisión binarios o N-arios.

```
[["¿Sos alto?", [{"SÍ", ["¿Te gustan los fuegos artificiales?", [{"SÍ", "Mago"}, {"No", "Elfo"}]}], [{"No", ["¿Te gusta el oro?", [{"SÍ", "Enano"}, {"No", "Hobbit"}]}]]]
```

- **Diccionarios:** Representan la jerarquía como claves con valores que también pueden ser diccionarios (subárboles) o valores finales. Son útiles cuando se desea acceder por clave de forma rápida.

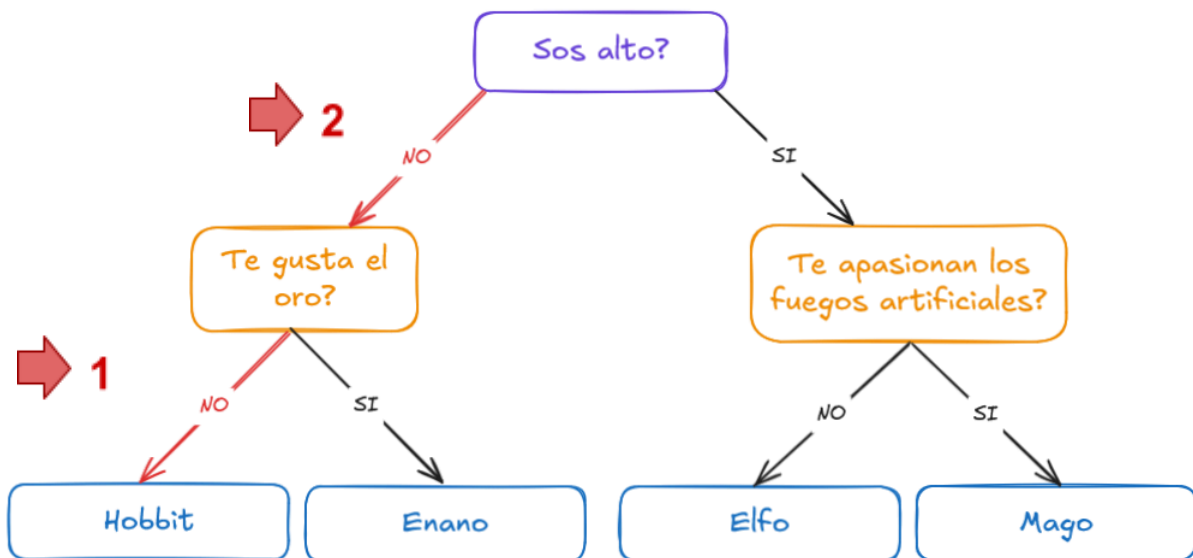
```
{
  "¿Sos alto?": {
    "SÍ": {
      "¿Te gustan los fuegos artificiales?": {
        "SÍ": "Mago",
        "No": "Elfo"
      }
    },
    "No": {
      "¿Te gusta el oro?": {
        "SÍ": "Enano",
        "No": "Hobbit"
      }
    }
  }
}
```

Ambas representaciones tienen sus ventajas. Las listas anidadas son más naturales para recorrer de manera secuencial o recursiva, mientras que los diccionarios ofrecen mayor eficiencia para accesos

por clave. Sin embargo, en ambos casos, se puede aplicar recursividad para recorrer o procesar el árbol completo.

2.1.iii Propiedades de los árboles

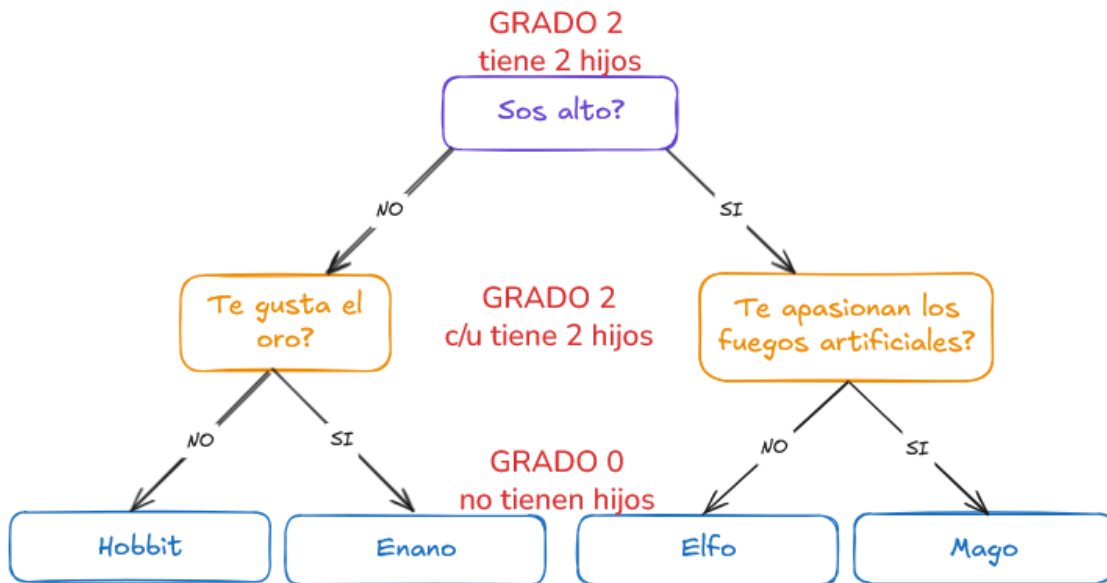
- **Longitud de camino:** Cuando deseas llegar de un nodo a otro dentro de un árbol debes seguir un recorrido llamado **camino**. Formalmente, un camino es una lista de ramas sucesivas que van de n_1 a n_2 . Estos recorridos se pueden realizar únicamente por las ramas de los árboles. La **longitud de un camino** es el número de ramas que hay que transitar para llegar de un nodo a otro.
- La **profundidad** de un nodo es la longitud de camino entre el nodo raíz y él mismo.



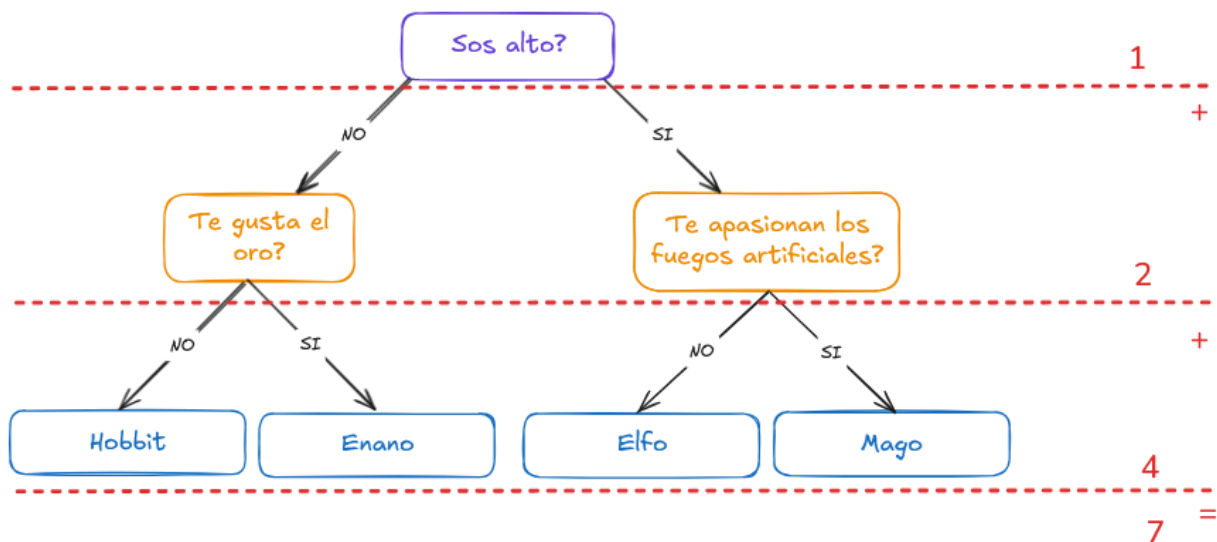
En la Figura :

- El nodo violeta tiene una profundidad igual a 0 ya que no se debe transitar ninguna rama para llegar a la raíz.
- Los nodos naranjas tienen una profundidad igual a 1 ya que se debe transitar una única rama en cada caso para llegar a la raíz.
- Los nodos azules tienen una profundidad igual a 2 ya que se deben transitar dos ramas en cada caso para llegar a la raíz.
- Además se visualiza un camino desde el nodo azul Hobbit hasta el nodo violeta o raíz; en este caso la longitud del camino es 2.

- El **nivel** de un nodo es la longitud del camino que lo conecta al nodo raíz más uno. Un nivel puede contener uno o más nodos. La diferencia entre nivel y profundidad es que el nivel incluye a la raíz mientras que la profundidad no la tiene en cuenta.
- La **altura** de un árbol es el máximo nivel del mismo. En el ejemplo explicado anteriormente la altura del árbol es igual a 3 ya que su nivel máximo es el 3.
- El **grado** de un nodo es el número de hijos que tiene dicho nodo. El grado de un árbol es el grado máximo de los nodos del árbol.



- El **orden** de un árbol es la máxima cantidad de hijos que puede tener cada nodo. A diferencia del grado, el orden es una cualidad que no se calcula una vez que el árbol ha sido construido sino que se establece como restricción antes de construirlo. Por ejemplo este árbol, al ser binario, **es un árbol de orden 2**
- Peso:** Es el número total de nodos que tiene un árbol. En programación es importante conocer el peso porque nos da una idea del tamaño del mismo y, por lo tanto, de la cantidad de memoria que podría ocupar, resulta crucial evaluar esto en aplicaciones donde la memoria es limitada como dispositivos móviles o sistemas embebidos. Además, un árbol de búsqueda con un peso demasiado alto podría presentar tiempos de ejecución lentos ya que recorrerlo tomaría más tiempo. Podemos utilizar esta información para balancearlo correctamente asegurándonos de que su altura sea lo más baja posible. **El peso de este árbol es 7**



2.1.iv Aplicaciones de los árboles

Los árboles son estructuras fundamentales en informática que se utilizan para organizar y gestionar datos de manera eficiente. Gracias a su estructura jerárquica, son ideales para representar relaciones padre-hijo y facilitar la navegación o búsqueda dentro de conjuntos complejos de información. Por eso, se aplican en diversas áreas, desde la gestión de archivos y bases de datos hasta la inteligencia artificial y el desarrollo de interfaces de usuario. Algunos ejemplos son:

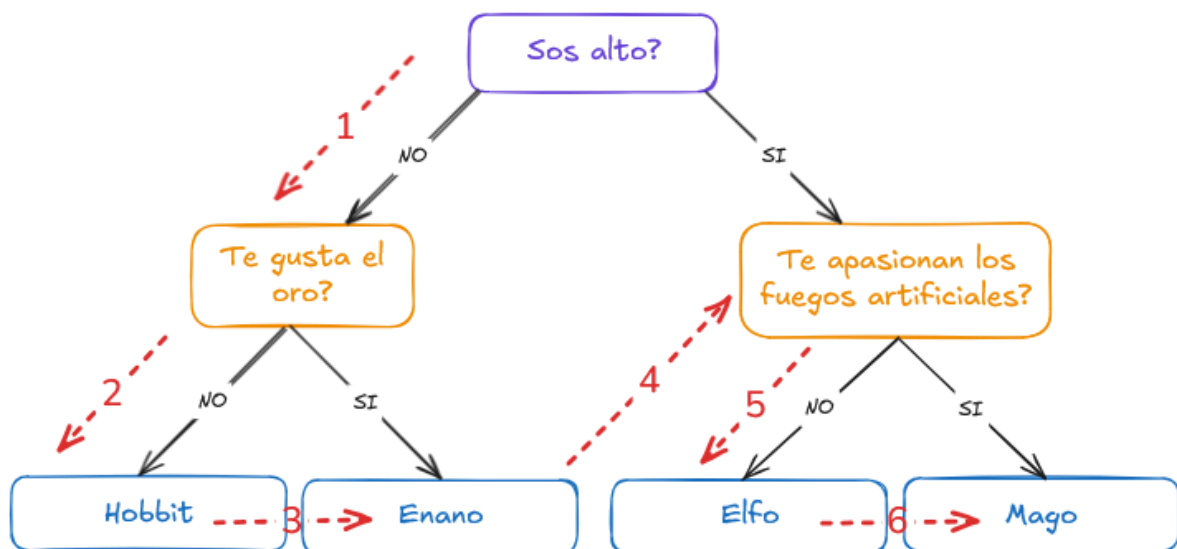
- Sistemas de archivos en computadoras
- Árboles genealógicos
- Menús de navegación en sitios web
- Árboles de decisión en inteligencia artificial
- Compiladores para análisis sintáctico
- Bases de datos y estructuras indexadas (como árboles B)
- Representación de expresiones matemáticas
- Juegos interactivos basados en elecciones
- Algoritmos de búsqueda y clasificación
- Redes de telecomunicaciones y enrutamiento

2.1 FORMA DE RECORRER LOS ÁRBOLES

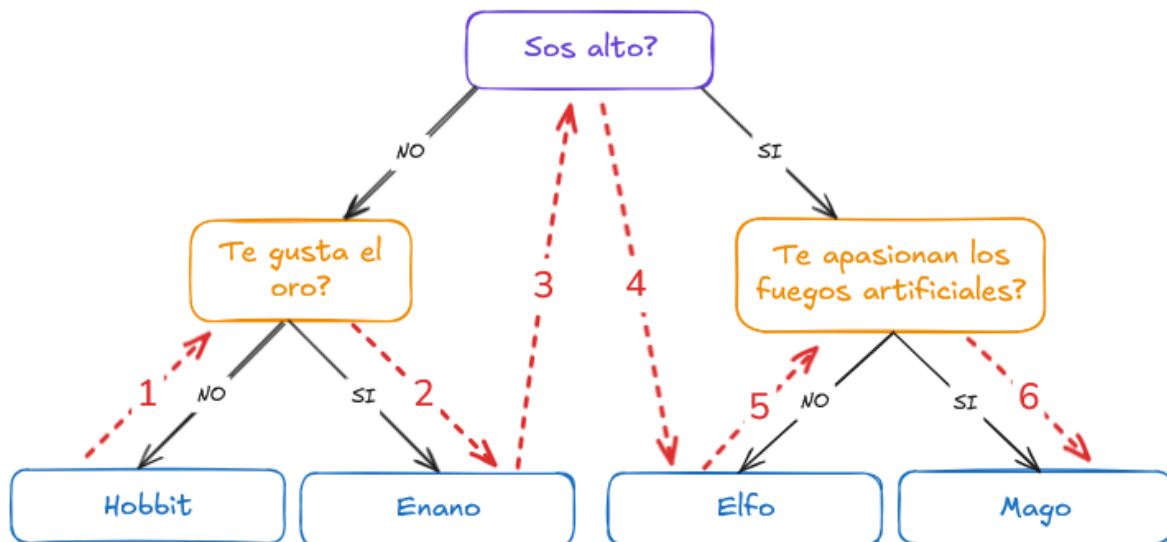
Existen varias formas de recorrer un árbol. Las tres formas más comunes son preorden, inorden y postorden.

- **Preorden:** El recorrido preorden es útil cuando necesitas realizar alguna operación en el nodo antes de procesar a sus hijos. Un caso típico es cuando estás copiando un árbol o clonando su estructura. Se ejecutan los siguientes pasos:

1. Se comienza por la raíz.
2. Se baja hacia el hijo izquierdo de la raíz.
3. Se recorre recursivamente el subárbol izquierdo.
4. Se sube hasta el hijo derecho de la raíz.
5. Se recorre recursivamente el subárbol derecho.

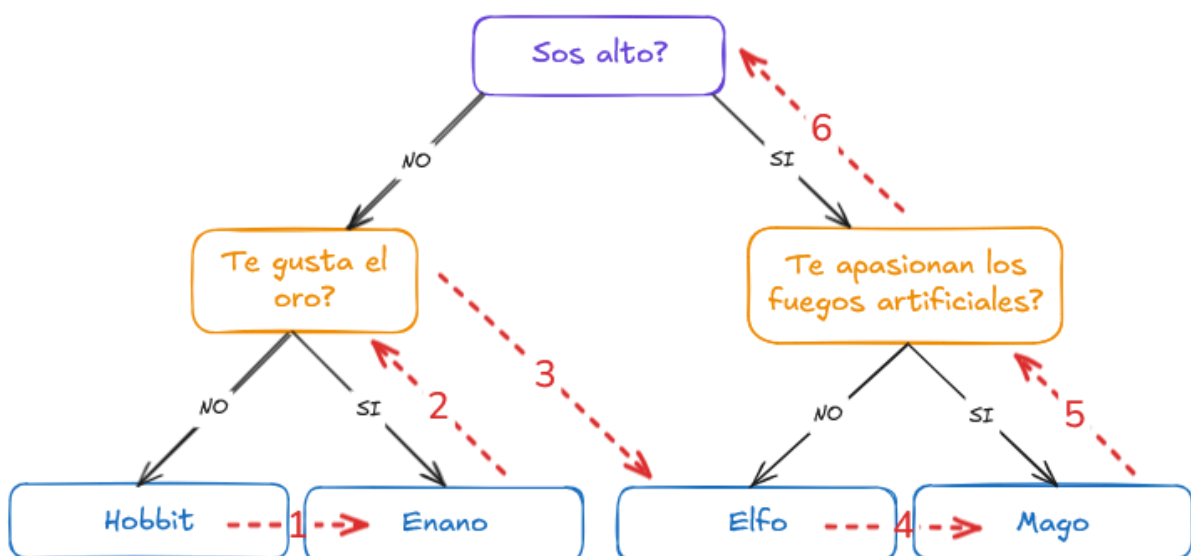


- **Inorden:** El recorrido inorden es especialmente útil para los árboles de búsqueda binaria, ya que garantiza que los nodos se visiten en orden ascendente. Se ejecutan los siguientes pasos:
1. Se comienza por el nodo hoja que se encuentre más a la izquierda de todos.
 2. Se sube hacia su nodo padre.
 3. Se baja hacia el hijo derecho del nodo recorrido en el paso 2.
 4. Se repiten los pasos 2 y 3 hasta terminar de recorrer el subárbol izquierdo.
 5. Se visita el nodo raíz.
 6. Se recorre el subárbol derecho de la misma manera que se recorrió el subárbol izquierdo.



- **Postorden:** El recorrido postorden es útil cuando quieres realizar alguna acción en los nodos hijos de un nodo antes de procesar el nodo mismo. Esto es común cuando se trabaja con estructuras jerárquicas donde necesitas procesar primero las subestructuras. Se ejecutan los siguientes pasos:

1. Se comienza por el nodo hoja que se encuentre más a la izquierda de todos.
2. Se visita su nodo hermano.
3. Se sube hacia el padre de ambos.
4. Si tuviera hermanos, se visita su nodo hermano.
5. Se repiten los pasos 3 y 4 hasta terminar de recorrer el subárbol izquierdo.
6. Se recorre el subárbol derecho, comenzando por el nodo hoja que se encuentre más a la izquierda y siguiendo el mismo procedimiento que con el subárbol izquierdo
7. Se visita el nodo raíz.



2.2.a Formas de recorrer un árbol: Recursividad e Iteración

Recursividad

La recursividad es una técnica que consiste en que una función se llame a sí misma para resolver un problema dividiéndolo en subproblemas más pequeños. Dado que cada nodo de un árbol puede ser considerado como la raíz de un subárbol, la recursividad encaja naturalmente con la estructura del árbol.

Este método permite implementar fácilmente los recorridos clásicos, como preorden, inorden y postorden, con código simple y claro. Sin embargo, la recursividad puede consumir más memoria debido al uso de la pila de llamadas, especialmente en árboles muy profundos, y puede llevar a un desbordamiento si no se maneja correctamente.

Iteración

La iteración recorre un árbol utilizando estructuras auxiliares para visitar los nodos sin hacer llamadas recursivas. La ventaja de la iteración es que el control sobre el uso de memoria y el flujo del programa es más explícito, lo que puede ser beneficioso para árboles grandes o cuando se quiere evitar los riesgos asociados con la recursividad profunda. Sin embargo, la implementación iterativa puede ser más compleja y menos intuitiva en comparación con la recursividad.

3. Caso práctico

En este trabajo desarrollamos dos implementaciones distintas de un árbol de decisiones para un juego simple donde permitimos que el usuario avance por distintas rutas según sus elecciones, utilizando el lenguaje Python. Una versión fue realizada con **diccionarios anidados**, representando cada nodo con sus posibles decisiones como claves. La otra fue implementada con **listas**, simulando la estructura jerárquica mediante sublistas organizadas por niveles. Probando además mediante funciones la obtención de propiedades características.

Con este trabajo buscamos resolver cómo modelar decisiones jerárquicas utilizando estructuras complejas (listas anidadas y diccionarios) en Python, sin recurrir a programación orientada a objetos, y enfocándose en la claridad del recorrido, la organización de los datos y la eficiencia en la navegación entre opciones.

3.1 IMPLEMENTACIÓN CON LISTAS

En esta parte se realizará el juego de averiguación de raza en un árbol compuesto por listas. Será un árbol binario donde hay 2 posibles respuestas que serán los hijos y las hojas serán las razas. En Python 3 utilizaremos, como se especifica en la teoría, una alternativa usando listas del tipo [valor, subárbol_izquierdo, subárbol_derecho], donde cada subárbol también es una lista similar.

Estructura base

Creemos el árbol como si fuera una lista, donde el índice 0 de la lista es la pregunta, el "SI" equivale a una rama con índice 0 de una sublista y el "NO" equivale a otra rama con índice 1. A cada una de esas respuestas se les adiciona una pregunta (con índice 1) donde volvemos a tener índices 0 y 1, pero esta vez, para las hojas.

```
arbol_razas = [
    "¿Sos alto?",
    [
        ["Si", [
            "¿Te apasionan los fuegos artificiales?",
            [
                ["Si", "Magos"],
                ["No", "Elfos"]
            ]
        ]],
        ["No", [
            "¿Te gusta el oro?",
            [
                ["Si", "Enanos"],
                ["No", "Hobbits"]
            ]
        ]]
    ]
]
```

Caso 1: Jugar

Creemos la función que permite jugar y recorrer ese árbol, les asigna un índice a los nodos, ya sean preguntas u opciones. Y mediante un condicional accede al lado derecho o al izquierdo del árbol. Primero se recorren las posibles respuestas; si hoja es una cadena (es una raza), se retorna o si hoja es otro nodo (una pregunta), el bucle continúa desde ahí.

Al finalizar se listan métodos de conversión de cadenas para ajustar las respuestas a los requerimientos mediante comentarios.

```
def jugar_razas(nodo):
    while True:
        pregunta, opciones = nodo
        respuesta = input(pregunta + " Responde con si o con no: ")
        respuesta = respuesta.strip().capitalize()
        for opcion, hoja in opciones:
```

```

        if respuesta == opcion:
            if type(hoja) == str: # validación directa del tipo
                return hoja
            else:
                nodo = hoja      # continuar recorriendo el árbol
                break
    else:
        print("Respuesta no válida. Usa 'Si' o 'No'.")

```

Así se ven las preguntas a medidas que ejecutamos código:

```

11.exe "f:/PROGRAMACION/0 UTN/REPOS/TP-GRUPAL-TUPAD-G1/integrador-programacion.py"
¿Sos alto? Responde con si o con no: si
¿Te apasionan los fuegos artificiales? Responde con si o con no: 

```

Caso 2: Calcular longitud

En esta parte comenzamos a listar las funciones de las propiedades, mi compañera realizó en el programa anterior el grado y el peso; así que en esta parte veremos la longitud del camino y la altura máxima. Siendo la primera la longitud de camino que toma la raza, es decir la hoja del árbol y cuenta hacia atrás el camino a la pregunta raíz. Si el nodo actual es una hoja, solo devuelve el nivel si coincide con destino, si no, devuelve -1 (indicando que no encontró el destino aquí).

Si el nodo es una pregunta; recorre las opciones, buscando recursivamente el destino en cada hoja, si alguna rama devuelve algo distinto de -1, significa que lo encontró y retorna el resultado. Si ninguna lo encontró, devuelve -1.

```

# Retorna la longitud de un nodo específico en el árbol.
def longitud_camino(nodo, destino, nivel=0):
    if type(nodo) == str:
        return nivel if nodo == destino else -1
    pregunta, opciones = nodo
    for opcion, hoja in opciones:
        res = longitud_camino(hoja, destino, nivel + 1)
        if res != -1:
            return res
    return -1

```

Caso 3: Calcular altura

Después se pasa la altura que no es más que el nivel máximo de profundidad, es decir, el nivel de un nodo es la longitud del camino que lo conecta al nodo raíz + 1. Un nivel puede contener uno o más nodos. La altura de un árbol es el máximo nivel del mismo. En el código si el nodo es una hoja, devuelve 1.

Si es una pregunta recorre las opciones y calcula recursivamente la altura de cada subárbol. Compara y se queda con la **mayor altura**. Al final, retorna esa altura +1

```

#Retorna la profundidad máxima del árbol.
def altura_max(nodo):
    if type(nodo) == str:

```

```

        return 1
    _, opciones = nodo
    alt = 0
    for opcion, hoja in opciones:
        sub_alt = altura_max(hoja)
        if sub_alt > alt:
            alt = sub_alt
    return alt + 1

```

Por último ejecutamos todas funciones para que se produzca un resultado

```

if __name__ == "__main__":
    raza = jugar_razas(arbol_razas)
    print(f"\n🧙‍♂️ Pertenece a la raza de los: {raza}")
    print(arteAscii(raza))

    camino = longitud_camino(arbol_razas, raza)
    alt = altura_max(arbol_razas)
    print(f"\nLongitud del camino hasta '{raza}': {camino}")
    print(f"Altura máxima del árbol: {alt}")

    imprimir_arbol(arbol_razas)

```

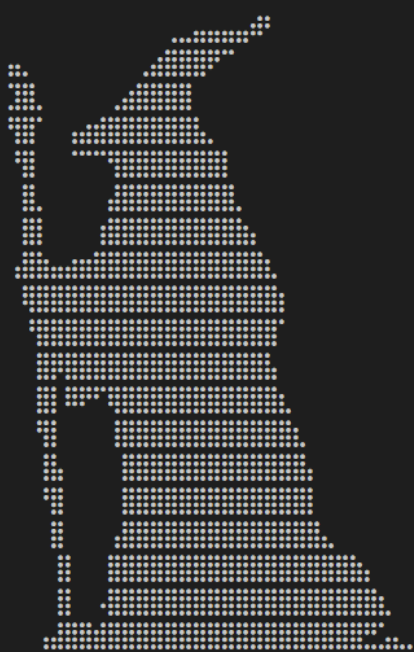
Los resultados que se obtienen completos son los siguientes:

```

● 11.exe "f:/PROGRAMACION/0 UTN/REPOS/TP-GRUPAL-TUPAD-G1/integrador-programacion.py"
¿Sos alto? Responde con si o con no: si
¿Te apasionan los fuegos artificiales? Responde con si o con no: si

🧙‍♂️ Pertenece a la raza de los: Magos

```



Longitud del camino hasta 'Magos': 2

Altura máxima del árbol: 3

¿Sos alto?

[Si]

¿Te apasionan los fuegos artificiales?

[Si]

- Magos

[No]

[No]

¿Te gusta el oro?

[Si]

- Enanos

[No]

- Hobbits

3.2 IMPLEMENTACIÓN CON DICCIONARIOS

Vamos a trabajar sobre un **árbol binario con diccionarios** donde podrán elegir entre diferentes preguntas con respuestas de decisión de si/no. Para este caso trabajamos con diccionarios, funciones y recursividad.

El objetivo final de este ejercicio es mediante una serie de preguntas básicas con respuestas si/no determinar a qué personaje ficticio pertenece cada usuario.

Estructura base

Creamos un diccionario (pares clave - valor), donde:

- Raíz: Pregunta inicial
- Ramas: Respuestas de si/no
- Hojas: Resultados finales (personaje)

```
arbol = {
    "pregunta": "¿Sos alto?",
    "si": {
        "pregunta": "¿Te apasionan los fuegos artificiales?",
        "si": {"resultado": "Sos un Mago"},
        "no": {"resultado": "Sos un Elfo"}
    },
    "no": {
        "pregunta": "¿Te gusta el oro?",
        "si": {"resultado": "Sos un Enano"},
        "no": {"resultado": "Sos un Hobbit"}
    }
}
```

Caso 1: Jugar

- Creamos una función donde recorreremos el árbol según respuestas del usuario donde:
- si llega al resultado final (hoja), muestra el personaje.
- si está en una pregunta (rama), espera la respuesta y continúa por el nodo [si/no] correspondiente.
- si ingresa una respuesta mal, lo llevamos a empezar de nuevo el juego.

```
def jugar(nodo):
    if "resultado" in nodo:
        print(f"Resultado: {nodo['resultado']}")
        return
    respuesta = input(nodo["pregunta"] + " (si/no): ").strip().lower()
    if respuesta == "si":
        jugar(nodo["si"])
    elif respuesta == "no":
        jugar(nodo["no"])
```

```

else:
    print("Respuesta no válida. Por favor respondé 'si' o 'no'.")
    jugar(nodo)

```

Ejecutamos la funcion jugar pasándole como argumento el árbol previamente creado

```
jugar(arbol)
```

Resultados:

```

MacBook-Pro de Maru:repos-marabujank@ /usr/local/bin/python3 /Users/marabujank/Desktop/repos
¿Sos alto? (si/no): si
¿Te apasionan los fuegos artificiales? (si/no): no
Resultado: Sos un Elfo
MacBook-Pro de Maru:repos-marabujank@ /usr/local/bin/python3 /Users/marabujank/Desktop/repos /cp-pro
¿Sos alto? (si/no): no
¿Te gusta el oro? (si/no): si
Resultado: Sos un Enano
MacBook-Pro de Maru:repos-marabujank@ /usr/local/bin/python3 /Users/marabujank/Desktop/repos /cp-pro

```

Caso 2: Agregar nodo

Realizamos una función para agregar nuevas preguntas en una hoja del árbol para poder hacerlo escalable, donde definimos estos parámetros:

- arbol
- camino: lista de pasos ("si"/"no") para llegar a la hoja
- nueva_pregunta: texto de la nueva pregunta
- resultado_si: resultado si la respuesta es "si"
- resultado_no: resultado si la respuesta es "no"

Validaciones:

- si el "resultado" no está en el nodo no se podrá agregar la pregunta ya que no estaríamos en una hoja sino en una rama.
- si se intenta insertar el nodo dentro de un camino no válido.
- al final se reemplazamos a la hoja definida por nodo nuevo completo

```

def agregarNodo(arbol, camino, nueva_pregunta, resultado_si, resultado_no):
    nodo = arbol
    for paso in camino:
        if paso in nodo:
            nodo = nodo[paso]
        else:
            print(f"Camino inválido: {paso} no existe")
            return
    if "resultado" not in nodo:
        print("No se puede agregar aquí porque este nodo ya tiene una pregunta.")
        return
    # Reemplazamos el nodo hoja por una nueva pregunta con dos resultados

```

```

nodo.clear()
nodo["pregunta"] = nueva_pregunta
nodo["si"] = {"resultado": resultado_si}
nodo["no"] = {"resultado": resultado_no}
print(f'Se agregó la pregunta "{nueva_pregunta}" al árbol')
print(f'El nodo completo: {nodo}')

```

- Ejecutamos la función jugar pasándole los argumentos basándose en la estructura del árbol.

```

print("Árbol actualizado:")
agregarNodo(
    arbol,
    camino=["si", "si"],
    nueva_pregunta="¿Tenés barba larga?",
    resultado_si="Sos un Mago",
    resultado_no="Sos un Hechicero joven"
)

```

- Resultados:

```

Árbol actualizado:
Se agregó la pregunta "¿Tenés barba larga?" al árbol
El nodo completo: {'pregunta': '¿Tenés barba larga?', 'si': {'resultado': 'Sos un Mago'}, 'no': {'resultado': 'Sos un Hechicero joven'}}

```

Caso 3: Calcular peso

- Calcular el peso del árbol: Creamos una función para obtener el número total de nodos lógicos que tiene el mismo.

```

# Peso: el número total de nodos
def calcularPeso(nodo):
    if isinstance(nodo, dict):
        if "resultado" in nodo:
            return 1
        suma = 1
        if "si" in nodo:
            suma += calcularPeso(nodo["si"])
        if "no" in nodo:
            suma += calcularPeso(nodo["no"])
        return suma
    return 0

```

- Ejecutamos el código

```
#Ejecutar codigo

peso = calcularPeso(arbol)
print(f"El peso del árbol es igual a {peso}.")
```

- Resultado:

```
El peso del árbol es igual a 7.
```

Caso 4: Calcular grado

- Calcular el grado del árbol: número de hijos que tiene dicho nodo

```
d# El grado es el número de hijos que tiene dicho nodo
def calcularGrado(nodo):
    if type(nodo) is dict:
        return len(nodo) - 1 # Restamos 1 porque no contamos la pregunta o
resultado
    return 0 # Si es una hoja, no tiene hijos
```

- Ejecutamos la función:

```
grado = calcularGrado(arbol)
print(f'El grado del árbol es: {grado}')
```

- Resultados:

```
El grado del árbol es: 2
```

CASOS EXTRAS

Decidimos colocarlos debido a que la implementación con listas, el primero suma una función que podría o no estar ya que el árbol binario realizado en listas es fácil de comprender. El segundo caso fue una decisión grupal en donde quisimos, ya que era un juego, mostrar algo más lúdico que solo el código con los resultados.

Para el caso de implementación con diccionarios, sumamos una función para saber cual es el recorrido que realiza una pregunta.

Con listas:

CASO 1:

Imprime el árbol binario, pero no nos pareció pertinente incluirlo en el código final ya que el árbol binario de listas se comprende bastante bien solo con observar al tener un estructura simple:

```
# Imprime el árbol de decisiones de forma estructurada
def imprimir_arbol(nodo, nivel=0):
    if type(nodo) == str:
        print(' ' * nivel + '- ' + nodo)
    else:
        pregunta, opciones = nodo
        print(' ' * nivel + pregunta)
        for opcion, subnodo in opciones:
            print(' ' * (nivel + 1) + f'[{opcion}]\n')
            imprimir_arbol(subnodo, nivel + 2)
```

CASO 2:

Este condicional imprime figuras dependiendo la raza, fue creado por razones lúdicas por lo que no es necesario integrarlo en el código.

[illegible]

[illegible]

```

        .===/_.'\\_/'-.\\===.
        .'\\ /          \\ / \'.
        /___|_\\          /_\\___\\_
        <___>'\\ \'.      ,' /<___>
        / / >==`-._.-'==< / /
        _/=== ' / ,---:---. \\/=, '
        / _ /      |_/v^v^v^v^\\_) \\
        \\| |)      |V^V^V^V^V^V|\\_/_/
        ||          \\|`-----|-----'/
        ||          \\|--._|_._,--/
        ||          |___|___|
        ||          <___X___>
        /A || A\\      \\|...|.../
        // \\| | / \\|\\      \\| | /
        (( [ ] ))      / v | v \\
        \\|\\ / \\ \\ //      / ,^ \\
        \\V      V/      `--'  `--'

    """)
elif raza == "Hobbits":
    return("""
        .dhh.
        .. hhhh.
        h: 'hhhhh.
        'h. .hhhhhh.
        'hhhhhhhhhh.
        'h?'hhhhhhhh'
        h. 'hhhhh'
        h. hhhhhh.
        h. hhhhhh?h
        h. hhhhhhhh.
        h.'h. 'hh.
        hhh..dh..hh..dh..hh..
        hhhhhhhhhhhhhhhhhhh.

    """)
)

```

Con diccionarios:

Caso 1: Realizamos una función para buscar el camino desde la raíz hasta una pregunta o resultado. Devuelve una lista con las preguntas y el resultado si lo encuentra, o None si no existe

```

def buscarCamino(nodo, destino, camino=None):
    if camino is None:
        camino = [] # Inicializamos el camino si es la primera llamada
    # Si el nodo es una pregunta, la agregamos al camino
    if "pregunta" in nodo:
        camino_actual = camino + [nodo["pregunta"]]

```



```

    if nodo["pregunta"] == destino:
        return camino_actual # Si encontramos la pregunta buscada
    # Buscamos recursivamente en las ramas 'si' y 'no'
    for respuesta in ["si", "no"]:
        if respuesta in nodo:
            resultado = buscarCamino(nodo[respuesta], destino, camino_actual)
            if resultado:
                return resultado

    # Si el nodo es un resultado (hoja), verificamos si es el destino
    if "resultado" in nodo:
        if nodo["resultado"] == destino:
            return camino + [nodo["resultado"]] # Devolvemos el camino completo
    # Si no se encontró el destino en este camino
    return None

```

- Ejecutamos la función, pasándole como argumento el árbol y una pregunta para buscar el recorrido que tenemos que hacer para llegar a esa pregunta.

```

print(buscarCamino(arbol, "¿Te apasionan los fuegos artificiales?")) # Debería
devolver el camino hasta esa pregunta

```

- Resultado: no muestra el camino que tuvo que hacer para llegar a ese recorrido

```

['¿Sos alto?', '¿Te apasionan los fuegos artificiales?']

```

4. Metodología Utilizada

La metodología que utilizamos para este trabajo integrador se basó en la colaboración constante donde podemos identificar las siguientes etapas:

Investigación previa:

Comenzamos con una búsqueda de información teórica sobre estructuras de datos avanzadas y sobre árboles, su funcionamiento y sus formas de representación en lenguajes de programación. Consultamos material de clase, videos de youtube, documentación oficial de Python y ejemplos de código disponibles en plataformas como W3Schools.

Diseño y desarrollo del código:

Definimos un caso práctico común: un árbol de decisiones aplicado a un juego simple. A partir de ese diseño conceptual donde plasmamos las ideas en un programa de diseño (excalidraw) pudimos tomar la decisión de que cada integrante implementara una versión diferente de la misma solución:

- Una versión con **listas anidadas**, que simula la jerarquía mediante posiciones.
Cada versión incluye un sistema de navegación paso a paso, que guía al usuario en función de sus elecciones.
- Y otra versión con **diccionarios anidados**, que permite nombrar explícitamente cada decisión.

Herramientas utilizadas:

Utilizamos el **IDE Visual Studio Code** para escribir y probar el código, con Python como lenguaje base. Donde nos ayudamos de jupyter (.ipynb) para poder ejecutar el código por bloques. Se realizaron pruebas locales en consola para verificar el funcionamiento y los recorridos del árbol. Para la organización del trabajo compartido, utilizamos **Google Drive** y para la presentación de diapositivas y edición del video utilizamos **Canva**. Para los encuentros sincrónicos utilizamos zoom, meet y discord, y para los asincrónicos whatsapp.

Trabajo colaborativo:

El trabajo fue realizado en grupo, donde cada una se encargó de desarrollar una versión distinta del árbol (una con listas, otra con diccionarios), lo que permitió comparar enfoques y evaluar sus ventajas y limitaciones. La redacción del informe y la organización del material se realizó en conjunto.

5. Resultados Obtenidos

El desarrollo del caso práctico nos permitió comprobar la funcionalidad de dos formas distintas de representar árboles de decisiones en Python, utilizando estructuras de datos avanzadas sin recurrir a programación orientada a objetos.

Pudimos comprobar que con ambas versiones, basado en listas y en diccionarios, logramos representar correctamente la lógica del árbol y permitir al usuario avanzar según sus decisiones. El flujo del juego se ejecuta de igual manera y permite llegar a diferentes finales en función de las elecciones realizadas.

Realizamos pruebas del recorrido completo en cada versión, seleccionando distintas combinaciones de respuestas para verificar que se llegaba al mismo resultado en todos los caminos posibles. También verificamos que no se produjeran errores si el usuario ingresaba una opción inválida.

Durante la fase de pruebas, detectamos errores relacionados con el control de opciones ingresadas por el usuario y con el acceso a nodos inexistentes (en particular en la versión con listas). Agregamos validaciones en ambos programas para evitar que se detuviera en caso de una entrada incorrecta.

Observamos que la versión con diccionarios es más legible e intuitiva a la hora de representar un árbol de decisiones, mientras que la versión con listas puede resultar más compacta pero requiere mayor cuidado en el manejo de las posiciones.

El código fuente de ambas versiones se encuentra disponible en el siguiente enlace:

<https://github.com/MaraBayurk/TUP-PROGRAMACION-INTEGRADOR>

6. Conclusiones

Con la ejecución de este trabajo pudimos entender cómo funcionan los árboles en estructura de datos, y cómo se pueden implementar en Python de diferentes formas. Pudimos ver en la práctica cómo una misma premisa puede resolverse con distintos enfoques, como listas o diccionarios, y qué ventajas o dificultades tiene cada una.

Uno de los desafíos fue validar correctamente las decisiones del usuario y evitar errores al acceder a elementos que no existían. Para eso, agregamos condiciones y mensajes que guiaban el flujo del programa sin que se corte.

Como mejora futura, nos gustaría hacer una versión usando clases para comprender y comparar cómo funcionan y llevarlo a la práctica.

Para finalizar consideramos que fue una experiencia muy positiva porque pudimos aplicarlo a un caso práctico y concreto, trabajando en equipo y probando distintas formas de resolver un mismo problema.

7. Bibliografía

- Documentación oficial de python:
<https://docs.python.org/es/3.13/tutorial/datastructures.html>
- Recursos subidos en el aula virtual como los documentos y videos de los profesores:
https://docs.google.com/document/d/10k16oL15EeyOaq92aoi4gwK3t_22X29-FSV2iV-8N1U/edit?tab=t.0#heading=h.d6ixjk6fxwp5
- https://www.w3schools.com/dsa/dsa_data_binarytrees.php
- <https://colab.research.google.com/drive/1yAGw7apt05qP9YAnFOPXF3mUYb2TYkY8?authuser=1#scrollTo=k9af3JolEvUZ>
- <https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/>

8. Link a video de youtube

<https://youtu.be/zrXtcT8cS1U>