Estructuras Jerárquicas en Python:

Arboles con listas anidada

Materia: Programación 1

Profesora: Cinthia Rigoni

Alumnos: Gonzalo Lujan, Sandra Loaiza

Fecha de entrega: 09/06/2025

Índice

1. Introducción

En el desarrollo de software y estructuras de datos, los árboles representan una herramienta fundamental para organizar información de manera jerárquica. Aunque existen estructuras especializadas para implementar árboles (como clases y objetos), también es posible representarlos usando listas anidadas, una técnica accesible y útil para quienes comienzan a programar en Python. Este trabajo explorará el uso de listas anidadas para construir, recorrer y manipular árboles, facilitando la comprensión de su estructura y comportamiento.

Elegir hacer un trabajo sobre árboles utilizando listas anidadas en Python permite explorar estructuras de datos fundamentales desde una perspectiva más didáctica y accesible. Aunque en la práctica se suelen usar clases para implementar árboles, utilizar listas anidadas es una excelente manera de comprender su estructura interna sin necesidad de conocimientos avanzados de programación orientada a objetos.

Además, este enfoque:

1. Simplifica el aprendizaje: Las listas anidadas permiten visualizar de forma clara la jerarquía de nodos y subnodos, lo cual facilita la comprensión del concepto de árbol.
2. Fomenta el pensamiento recursivo: Trabajar con listas anidadas obliga a utilizar funciones recursivas para recorrer o modificar el árbol, lo cual es una habilidad clave en programación.
3. Es útil en contextos donde no se puede usar clases: En algunas situaciones educativas o entornos limitados, puede no estar permitido usar clases. Las listas anidadas son una alternativa válida y poderosa.
4. Permite abordar operaciones básicas: Como insertar nodos, recorrer el árbol (preorden, inorden, postorden), buscar valores, etc., lo cual cubre los objetivos fundamentales de un trabajo sobre árboles.

Los árboles son una de las estructuras de datos más importantes y versátiles en la programación. Su importancia radica en que permiten representar y organizar información jerárquica de forma eficiente de la siguiente manera:

Representan estructuras jerárquicas

Muchos problemas del mundo real tienen una estructura jerárquica (como carpetas dentro de carpetas, menús, o árboles genealógicos), y los árboles permiten modelarlos de manera natural.

Facilitan búsquedas eficientes

Estructuras como los árboles binarios de búsqueda (BST) permiten buscar datos de forma mucho más rápida que con listas normales, especialmente cuando hay muchos elementos.

Son base de otras estructuras

Estructuras más complejas como **heaps**, **tries**, y **árboles AVL** (tipos de específicos de árboles) se basan en la idea de árboles. Además, muchos algoritmos como los de ordenamiento, recorrido, e incluso algunos relacionados con inteligencia artificial usan árboles.

Aplicaciones reales

Los árboles se utilizan en:

- Motores de búsqueda (árboles de decisión, árboles de índice)
- Compiladores (árboles de sintaxis)
- Bases de datos (índices en forma de B-trees)
- Navegación de archivos y sistemas de carpetas
- Inteligencia artificial (como árboles de decisión y minimax en juegos)

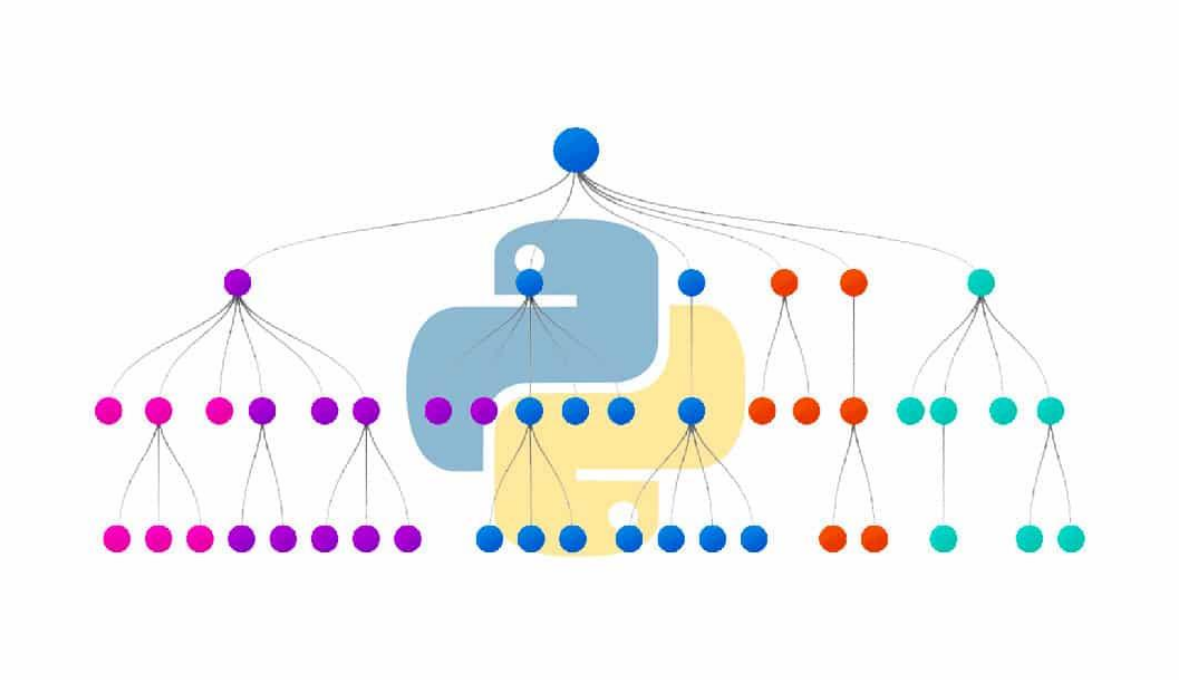
Desarrollan el pensamiento recursivo

Trabajar con árboles (especialmente con listas anidadas) ayuda a entender y aplicar la recursión, una herramienta muy poderosa en programación.

En resumen, estudiar árboles no solo es útil para resolver problemas específicos, sino también para mejorar el pensamiento lógico y prepararse para problemas más complejos en ciencia de datos, desarrollo de software y algoritmos avanzados.

2. Objetivos específicos

1. Se busca entender la estructura de un árbol y cómo puede representarse con listas anidadas en lugar de clases.
2. Implementar funciones básicas sobre árboles, como insertar elementos, recorrer el árbol y buscar valores.
3. Aplicar recursión para trabajar con estructuras de datos anidadas.
4. Desarrollar el pensamiento lógico y estructurado, mediante la solución de problemas usando árboles.
5. Relacionar los árboles con situaciones reales, como la organización de carpetas, decisiones lógicas o jerarquías.
6. Practicar el uso de listas en Python, fortaleciendo la base para estructuras más complejas.



3. Marco teórico

Estructuras de datos

En programación, una **estructura de datos** es una forma de organizar y almacenar información para que pueda ser utilizada de manera eficiente. Algunas de las estructuras más básicas son listas, pilas, colas y diccionarios. A medida que se avanza, se estudian estructuras más complejas como los árboles y los grafos, que permiten representar datos jerárquicos o conectados de una manera más lógica y útil.

Árboles

Un **árbol** es una estructura de datos no lineal que se compone de elementos llamados **nodos**. El nodo superior se conoce como **raíz** y desde él se ramifican otros nodos llamados **hijos**, formando así una jerarquía. Cada nodo puede tener varios hijos, pero solo tiene un padre (excepto la raíz, que no tiene ninguno).

Algunas características importantes de los árboles:

- Un árbol no tiene ciclos.
- La estructura es recursiva: cada subnodo puede considerarse a su vez la raíz de un subárbol.
- Permiten representar relaciones jerárquicas como carpetas en un sistema operativo, árboles genealógicos o decisiones lógicas.

Conceptos claves:

- **Raíz**: el nodo principal del árbol.
- **Padre e hijo**: relación entre nodos conectados directamente.
- **Hoja**: nodo sin hijos.
- **Subárbol**: árbol contenido dentro de otro nodo.
- **Recorrido**: proceso de visitar todos los nodos (preorden, inorden, postorden).

Árboles con listas anidadas en Python:

Una lista anidada es una lista que contiene otras listas. Esto permite crear estructuras similares a árboles, ya que cada lista puede representar un nodo y sus hijos. Usar listas anidadas permite representar un árbol donde cada nodo es una lista que contiene:

1. El valor del nodo.
2. Una lista con sus subárboles.

Ejemplo:

```
árbol = ['A',
        ['B',
          ['D', [], []],
          ['E', [], []]
        ],
        ['C',
          ['F', [], []],
          ['G', [], []]
        ]
       ]
```

En este ejemplo:

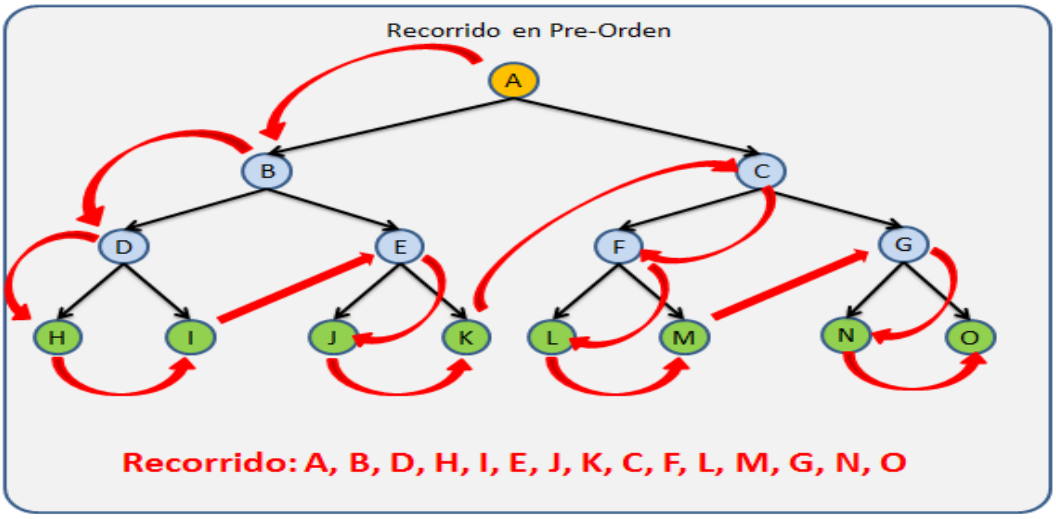
- 'A' es la raíz.
- 'B' y 'C' son hijos de 'A'.
- 'D' y 'E' son hijos de 'B', etc.

Cada nodo está representado como una lista de tres elementos: [valor, subárbol_izquierdo, subárbol_derecho].

Recorridos de árboles

Los **recorridos** son formas de visitar todos los nodos del árbol en cierto orden. Los más comunes son:

- **Preorden**: primero se visita el nodo actual, luego el subárbol izquierdo y después el derecho.



- **Inorden**: primero se visita el subárbol izquierdo, luego el nodo actual, y después el subárbol derecho.
- **Postorden**: primero se visitan los subárboles izquierdo y derecho, y al final el nodo actual.

Estos recorridos suelen implementarse con **funciones recursivas**, una técnica fundamental en programación.

Importancia de los árboles en programación

Los árboles son utilizados en múltiples aplicaciones, como:

- Organización de datos jerárquicos (sistemas de archivos)
- Búsqueda rápida (árboles binarios de búsqueda)
- Procesamiento de expresiones matemáticas
- Inteligencia artificial (árboles de decisión)
- Bases de datos (índices y estructuras como B-trees)

Estudiar árboles permite desarrollar un pensamiento lógico más estructurado y aprender herramientas clave como la recursión y la manipulación eficiente de datos.

3.Caso **práctico:**

Simulación un sistema de directorios y archivos usando listas anidadas en una estructura árbol, permitiendo:

1. Crear un árbol de directorios.
2. Agregar archivos o directorios
3. Buscar un archivo (por preorden)
4. Mostrar la estructura completa del árbol

Código:

```
# Agrega un directorio o archivo a la estructura del árbol (en lista anidada).
def agregar_directorio_o_archivo(arbol, ruta, tipo):
    if not ruta:
        return
    nombre = ruta[0]


# Se agrega un archivo a una lista que representa un directorio, pero solo si el archivo aún no existe.
    if len(ruta) == 1 and tipo == 'archivo':
        if nombre not in arbol:
            arbol.append(nombre)
        else:
            print(f"El archivo '{nombre}' ya existe en este directorio.")
        return


# Buscar si ya existe un directorio con ese nombre, si lo encuentra sigue agregando los niveles restantes de la ruta forma recursiva
    for item in arbol:
        if isinstance(item, list) and item[0] == nombre:
            agregar_directorio_o_archivo(item[1], ruta[1:], tipo)
            return


 # Si no existe, crear nuevo directorio
    nuevo_directorio = [nombre, []]
    arbol.append(nuevo_directorio)
    agregar_directorio_o_archivo(nuevo_directorio[1], ruta[1:], tipo)
```

```
#Busqueda en preorden de un archivo determinado
def buscar_archivo_preorden(arbol, nombre_archivo):

    for item in arbol:
        if isinstance(item, list):
            resultado = buscar_archivo_preorden(item[1], nombre_archivo)
            if resultado:
                return resultado
        elif item == nombre_archivo:
            return item
    return None

# Solicita una ruta y tipo al usuario.
def ingresar_directorio_y_archivo():
    ruta = input("Ingresa la ruta (ejemplo: home/user/docs/archivo.txt): ").strip()
    ruta_split = ruta.split('/')
    tipo = input("¿Es un directorio o un archivo? (directorio/archivo): ").strip().lower()
    return ruta_split, tipo


======================Programa principal ========================

arbol = [] #Se crea el árbol principal como una lista vacía.


# Se Agregan  elementos al árbol
while True:
    ruta, tipo = ingresar_directorio_y_archivo()
agregar_directorio_o_archivo(arbol, ruta, tipo)
    continuar = input("¿Deseas agregar otro directorio/archivo? (s/n):
").strip().lower()
    if continuar != 's':
        break

#Funcion para imprimir el arbol(forma jerarquica)
def imprimir_arbol(arbol, nivel=0)
for item in arbol:
 if isinstance(item, list):
            print("  " * nivel + f"[{item[0]}]")
            imprimir_arbol(item[1], nivel + 1)
        else:
            print("  " * nivel + f"- {item}")

# Búsqueda de archivo
```

```
nombre_a_buscar = input("Ingresa el nombre del archivo a buscar: ").strip()
resultado = buscar_archivo_preorden(arbol, nombre_a_buscar)

# Mostar si se encontro el archivo y imprimir el arbol
if resultado:
    print(f"Archivo encontrado: {resultado}" )
else:
    print(f"Archivo '{nombre_a_buscar}' no encontrado.")

imprimir_arbol(arbol)
```

Ejemplo de aplicación:

```
PS C:\Users\hp\Documents\GitHub\Codigo de PROGRAMACION 1> python -u
"c:\Users\hp\Documents\GitHub\Codigo de PROGRAMACION
1\codigo_trabajo_integracion.py"
Ingresa la ruta (ejemplo: home/user/docs/archivo.txt): home/user/docs/archivo.txt
¿Es un directorio o un archivo? (directorio/archivo): archivo
¿Deseas agregar otro directorio/archivo? (s/n): s
¿Es un directorio o un archivo? (directorio/archivo): archivo
¿Deseas agregar otro directorio/archivo? (s/n): n
Ingresa el nombre del archivo a buscar: archivo.txt
Archivo encontrado: archivo.txt
[home]
  [user]
    [docs]
      - archivo.txt
    [word]
      - archivo1.txt
```

Descripción del problema:

Queremos simular un **sistema de directorios o menús** similar al de un sistema operativo o una aplicación. Cada directorio puede tener subdirectorios, y estos a su vez pueden tener más un subdirectorio o archivo. Usando árboles representados con listas anidadas, se puede construir esta jerarquía y permitir:

- Agregar nuevos directories o archivo
- Buscar una archives por su nombre
- Mostrar toda la estructura jerárquica

Objetivo: Construir y recorrer un árbol representado con listas anidadas.

```
# Árbol que representa un sistema de carpetas
carpetas = [
    "raíz",
    ["Documentos",
        ["Tareas", [], []],
        ["Exámenes", [], []]
    ],
    ["Imágenes",
        ["Vacaciones", [], []],
        []
    ]
]
```

Decisiones de diseño del código

1. Representación del árbol con listas anidadas

Se decidió representar cada nodo como una lista de tres elementos:
[valor, subárbol_izquierdo, subárbol_derecho]

Esto facilita la organización de los datos jerárquicos de forma recursiva sin usar clases, y permite recorrer fácilmente la estructura usando funciones.

Ejemplo:

```
["raíz",
  ["Documentos", [], []],
  ["Imágenes", [], []]
]
```

2. Estructura recursiva para recorrer y modificar el árbol

Para tareas como insertar o buscar nodos, se utiliza recursión, ya que cada subárbol es una lista con la misma estructura. La recursión permite procesar árboles de cualquier profundidad sin necesidad de usar bucles complejos.

3. Separación de funcionalidades en funciones

Para mantener el código ordenado y reutilizable, se definieron funciones específicas para cada tarea:

buscar_nodo(arbol, nombre: Esta función busca un nodo por nombre dentro del árbol. Devuelve la sublista que representa ese nodo (si existe).

- insertar_nodo(arbol, nombre_padre, nuevo_nodo):Inserta un nuevo nodo dentro del nodo con nombre **nombre_padre**, si se encuentra.
- mostrar_arbol(arbol, nivel=0): Muestra todo el árbol con indentación según el nivel para visualizar su estructura.

Esto sigue el principio de modularidad, que consiste en dividir el programa en partes pequeñas que se pueden probar y modificar de forma independiente.

4. Uso de impresión recursiva para mostrar la jerarquía

Se diseñó una función mostrar_arbol() que imprime la estructura jerárquica con sangrados para visualizar la profundidad de cada nodo. Esto ayuda al usuario a entender la forma del árbol sin ver directamente la lista anidada.

5. Limitación a árboles binarios (2 hijos por nodo)

Aunque un árbol puede tener muchos hijos, se optó por un árbol **binario** (cada nodo tiene hasta 2 hijos) para simplificar la lógica. Esto hace que el diseño sea más fácil de manejar al principio y permite enfocarse en los conceptos fundamentales.

Resultado final esperado:

Un programa que le permita al usuario ver una estructura de carpetas organizada en forma de árbol, agregar nuevas carpetas en lugares específicos y buscar una carpeta para ver si existe.

5. Metodología **utilizada**

Investigación teórica

Se comenzó con la recopilación de información teórica sobre estructuras de datos, específicamente árboles, su utilidad, tipos y formas de implementación. Se puso especial énfasis en representar árboles de forma simple, usando listas anidadas en lugar de clases con el material propuesto por la cátedra.

Diseño del árbol

Se decidió representar cada nodo del árbol como una lista de tres elementos: [valor, subárbol_izquierdo, subárbol_derecho], siguiendo una estructura de árbol binario. Este diseño fue elegido por su claridad y facilidad de implementación en Python.

Desarrollo de funciones

Se implementaron funciones modulares para realizar operaciones comunes sobre árboles:

- insertar_nodo(): para agregar nuevos nodos al árbol.
- buscar_nodo(): para localizar un nodo específico por su nombre.
- mostrar_arbol(): para imprimir la estructura jerárquica del árbol con sangrados.

Estas funciones fueron diseñadas utilizando recursión, ya que es una técnica ideal para trabajar con estructuras anidadas como los árboles.

Pruebas del programa

Se realizaron pruebas de funcionamiento agregando datos al árbol, buscándolos y mostrando la estructura. Estas pruebas ayudaron a comprobar que el programa respondía correctamente a las operaciones y respetaba la estructura lógica del árbol.

Documentación del código y resultados

El código fue comentado de forma clara para facilitar su lectura y comprensión. Además, se elaboró documentación escrita que explica las decisiones tomadas, los objetivos y los resultados obtenidos con el desarrollo del trabajo.

6. Resultados **obtenidos**

A través del caso práctico propuesto, se logró:

- Representar correctamente un árbol con listas anidadas.
- Desarrollar una función recursiva para recorrer el árbol en preorden.
- Entender la jerarquía de nodos y la relación padre-hijo en árboles.
- Observar los beneficios de usar listas anidadas como una forma inicial de manipular estructuras de datos complejas en Python.

7. Conclusiones

El uso de listas anidadas para representar árboles en Python es una estrategia efectiva para introducir a los estudiantes en estructuras de datos complejas sin necesidad de programación orientada a objetos.

El ejercicio permitió visualizar la estructura jerárquica de los árboles y aplicar técnicas de recorrido, habilidades fundamentales en algoritmos y ciencias de la computación.

8. Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Downey, A. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.
- Python Software Foundation. (2024). *Python documentation*. https://docs.python.org/3/
- Material brindada por la cátedra.
- Normas APAhttps://normas-apa.org/formato/#google_vignette.