LECTURA SESIÓN 1: ESTRUCTURAS DE DATOS AVANZADAS

En el mundo de la programación y el desarrollo de software, las estructuras de datos son el pilar fundamental para organizar y manipular información de manera eficiente. Más allá de las estructuras básicas como listas y diccionarios, existen otras —consideradas "avanzadas"— que ofrecen soluciones óptimas a problemas más complejos, facilitando tanto el diseño de algoritmos como la implementación de sistemas de mayor escala.

por Kibernum Capacitación S.A.

¿Qué son las Estructuras de Datos Avanzadas?

Las estructuras de datos avanzadas incluyen, por ejemplo, pilas, colas, árboles, grafos, montículos (heaps) y otras variantes que permiten:

- Reducir la complejidad de ciertas operaciones (búsqueda, inserción, eliminación).
- Modelar problemas de manera más intuitiva (por ejemplo, árboles de decisión, redes de transporte, relaciones en redes sociales, etc.).
- Aumentar la legibilidad y mantenibilidad del código al representar de forma clara las relaciones entre los datos.

En esencia, cada estructura de datos se diseña para un propósito específico y brinda un conjunto de operaciones que la hace más o menos adecuada según el escenario. Entender en profundidad estas estructuras posibilita a los programadores abordar problemas de mediana o alta complejidad de forma más eficiente y elegante.

Aplicación en la Resolución de Problemas Complejos

A medida que los proyectos crecen en tamaño y complejidad, las estructuras de datos avanzadas se vuelven imprescindibles. Pensemos en un sistema de recomendación (típico en plataformas de streaming o comercio electrónico): el modelado de relaciones entre usuarios y productos puede requerir grafos para representar conexiones y preferencias, así como colas o pilas para gestionar los procesos de actualización de forma ordenada.

En el ámbito de la Inteligencia Artificial y el Aprendizaje de Máquina, estas estructuras permiten:

- Manejar de forma más eficiente grandes volúmenes de datos.
- Diseñar algoritmos de búsqueda o de recorrido (por ejemplo, en árboles de decisión).
- Optimizar flujos de datos (pipelines) donde se requiere un orden específico de procesamiento.

La elección de la estructura de datos adecuada puede impactar drásticamente el rendimiento, la escalabilidad y la calidad de los resultados obtenidos.

PYTHON Y LAS ESTRUCTURAS DE DATOS

Python es un lenguaje muy popular en la comunidad de machine learning y data science por múltiples razones:

1 Simplicidad y Legibilidad

Su sintaxis clara facilita la comprensión de conceptos avanzados sin distraerse con detalles excesivos de implementación.

2 Amplio Ecosistema de Librerías

Además de las estructuras de datos nativas (listas, tuplas, diccionarios, conjuntos), Python cuenta con módulos especializados como collections (que provee deque, Counter, defaultdict, etc.), heapq (para colas de prioridad), y librerías de terceros como networkx (para el manejo de grafos complejos), entre muchas otras.

3 Gran Comunidad y Documentación

La enorme base de usuarios y desarrolladores de Python garantiza abundantes recursos de aprendizaje, foros de soporte y soluciones compartidas, lo que agiliza la resolución de dudas y la experimentación con nuevas ideas.

4 Facilidad de Integración

Python se integra de forma muy sencilla con librerías escritas en C/C++ o con entornos de big data, lo que permite aprovechar lo mejor de ambos mundos: rapidez de ejecución y facilidad de programación.

Estas cualidades hacen que Python sea un lenguaje idóneo para la implementación de estructuras de datos avanzadas y para la construcción de prototipos de algoritmos de machine learning que puedan escalar a aplicaciones de producción.

Resumen de Estructuras de Datos en Python

En resumen, el estudio de estructuras de datos avanzadas es esencial para afrontar con éxito proyectos complejos, especialmente en áreas como el aprendizaje automático, donde la forma en que se organizan los datos puede determinar la viabilidad y la eficiencia de un modelo. Python, con su legibilidad, su extenso ecosistema y su comunidad activa, se presenta como una herramienta poderosa para aprender, experimentar e implementar estas estructuras de datos, sentando las bases para resolver problemas de mediana y alta complejidad de manera efectiva.

A continuación, revisaremos las estructuras de datos avanzadas y su aplicación:

1. Listas, Tuplas, Diccionarios y Conjuntos (Sets)
2. Pilas (Stacks) y Colas (Queues)
3. Árboles (Trees)
4. Grafos (Graphs)
5. Casos y Ejemplos de Uso Avanzado en Python

1. LISTAS, TUPLAS, DICCIONARIOS Y CONJUNTOS

1.1. Listas

Una lista en Python es una secuencia ordenada de elementos que puede contener datos de diferentes tipos (int, float, str, objetos, etc.).

Características:

1. Se definen con corchetes [ ].
2. Son mutables, es decir, se pueden agregar, modificar o eliminar elementos después de su creación.
3. Soportan indexación y slicing.

¿Qué es la indexación y el slicing en Listas?

Las listas en Python son colecciones ordenadas, lo que significa que cada elemento tiene una posición determinada dentro de la lista. Esta posición se denomina índice (index), y es la forma en que Python nos permite acceder o manipular los elementos de la lista.

1. Indexación

- Cada elemento de la lista se asocia a un número entero, empezando en 0 para el primer elemento, 1 para el segundo, y así sucesivamente.
- Podemos acceder a un elemento escribiendo el nombre de la lista seguido de corchetes con el índice correspondiente.
- Python también soporta índices negativos, donde -1 se refiere al último elemento, -2 al penúltimo, y así sucesivamente.

Ejemplo:

```
numeros = [10, 20, 30, 40, 50]

print(numeros[0])   # Imprime 10 (primer elemento)
print(numeros[3])   # Imprime 40 (cuarto elemento)
print(numeros[-1])  # Imprime 50 (último elemento)
```

```
10
40
50
```

2. Slicing (rebanado)

- El slicing permite extraer una porción (sublista) de la lista original, especificando un rango de índices.
- La sintaxis básica es: lista[inicio:fin], donde:
  - inicio es el índice del primer elemento que queremos incluir en la sublista.
  - fin es el índice hasta donde queremos llegar, pero no se incluye ese índice en la sublista.
- Opcionalmente, podemos especificar un paso (step) con lista[inicio:fin:paso].

Ejemplo:

```
numeros = [10, 20, 30, 40, 50]

print(numeros[1:4])    # Imprime [20, 30, 40] (desde índice 1 hasta índice 3)
print(numeros[:3])     # Imprime [10, 20, 30] (desde inicio hasta índice 2)
print(numeros[2:])     # Imprime [30, 40, 50] (desde índice 2 hasta el final)
print(numeros[::2])    # Imprime [10, 30, 50] (todos los elementos con paso 2)
```

```
[20, 30, 40]
[10, 20, 30]
[30, 40, 50]
[10, 30, 50]
```

Esta funcionalidad de indexación y slicing facilita el acceso, manipulación y extracción de subconjuntos de datos, siendo una de las características más potentes de las listas en Python.

Ejemplos de creación y manipulación de listas

Ejemplo 1: Creación y acceso a elementos de una lista

```
mi_lista = [10, 20, 30, 40, 50]

print(mi_lista[0])   # Accede al primer elemento (10)
print(mi_lista[-1])  # Accede al último elemento (50)
```

```
10
50
```

Ejemplo 2: Modificación de elementos

```
mi_lista[1] = 25

print(mi_lista)  # [10, 25, 30, 40, 50]
```

```
[10, 25, 30, 40, 50]
```

Ejemplo 3: Agregar y eliminar elementos

```
mi_lista.append(60)

print(mi_lista)  # [10, 25, 30, 40, 50, 60]
mi_lista.remove(30)
print(mi_lista)  # [10, 25, 40, 50, 60]
```

```
[10, 25, 30, 40, 50, 60]
[10, 25, 40, 50, 60]
```

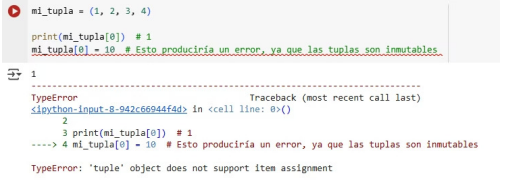
1.2. Tuplas

Una tupla es similar a una lista, pero sus elementos son inmutables, es decir, no se pueden modificar después de su creación.

Características

Se definen con paréntesis ( ).

Ejemplo de creación y acceso a elementos de tupla



```
mi_tupla = (1, 2, 3, 4)

print(mi_tupla[0])  # 1
mi_tupla[0] = 10  # Esto produciría un error, ya que las tuplas son inmutables
```

```
1
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-942c66944f4d> in <cell line: 0>()
      2
      3 print(mi_tupla[0])  # 1
----> 4 mi_tupla[0] = 10  # Esto produciría un error, ya que las tuplas son inmutables

TypeError: 'tuple' object does not support item assignment
```

Ejemplo de creación y acceso a elementos de tupla:

Ventajas

Son más ligeras en memoria y ofrecen cierta optimización en el acceso.

Uso común

Se usan para agrupar datos que no se espera que cambien.

1.3. Diccionarios

Estructura de datos que almacena pares clave-valor.

Definición

Se definen con llaves { }.

Operaciones

Permiten acceso, modificación y eliminación de elementos mediante la clave.

Aplicaciones

Son muy útiles para representar datos estructurados, como JSON.

Ejemplo de creación y manipulación de diccionarios

```python
mi_diccionario = {
    "nombre": "Alice",
    "edad": 30,
    "ciudad": "Madrid"
}

# Acceso a valores
print(mi_diccionario["nombre"])  # "Alice"

# Agregar o modificar un valor
mi_diccionario["profesion"] = "Ingeniera"
mi_diccionario["edad"] = 31

# Eliminar un par clave-valor
del mi_diccionario["ciudad"]
print(mi_diccionario)

# {"nombre": "Alice", "edad": 31, "profesion": "Ingeniera"}
```

```
Alice
{'nombre': 'Alice', 'edad': 31, 'profesion': 'Ingeniera'}
```

1.4. Conjuntos (Sets)

Colección no ordenada de elementos únicos (sin duplicados).

Definición

Se definen con llaves { } o con la función set().

Operaciones

Permiten operaciones matemáticas como unión, intersección, diferencia.

Aplicaciones

Son muy útiles para filtrar duplicados y para operaciones de pertenencia (membership).

Ejemplo de creación y operaciones con conjuntos

2. PILAS (STACKS) Y COLAS (QUEUES)

2.1. Pilas (Stacks)

Estructura de datos LIFO (Last In, First Out), donde el último elemento en entrar es el primero en salir.

Uso en Python:

- Se puede implementar usando una lista, empleando append() para apilar (push) y pop() para desapilar (pop).

Ejemplo:

```python
pila = []
pila.append("A")  # push
pila.append("B")
pila.append("C")

print(pila)       # ["A", "B", "C"]

ultimo = pila.pop()

print(ultimo)     # "C" (último en entrar, primero en salir)
print(pila)       # ["A", "B"]
```

```
['A', 'B', 'C']
C
['A', 'B']
```

2.2 Colas (Queues)

Estructura de datos **FIFO** (First In, First Out), donde el primer elemento en entrar es el primero en salir.

- **Uso en Python**:
- Se puede implementar con **collections.deque**, que es más eficiente para operaciones de cola.

Ejemplo

3. ÁRBOLES (TREES)

Un árbol es una estructura de datos jerárquica compuesta por nodos. Cada nodo puede tener nodos "hijo", y existe un único nodo raíz (root).

Conceptos Básicos

- Nodo raíz (root): el nodo superior que no tiene padre.
- Hojas (leaves): nodos sin hijos.
- Subárboles: cada nodo puede ser la raíz de su propio subárbol.

Dentro del estudio de los árboles, existen diversas clasificaciones que permiten optimizar y adaptar la estructura a distintos tipos de problemas. A continuación, se describen algunos de los tipos de árboles más relevantes:

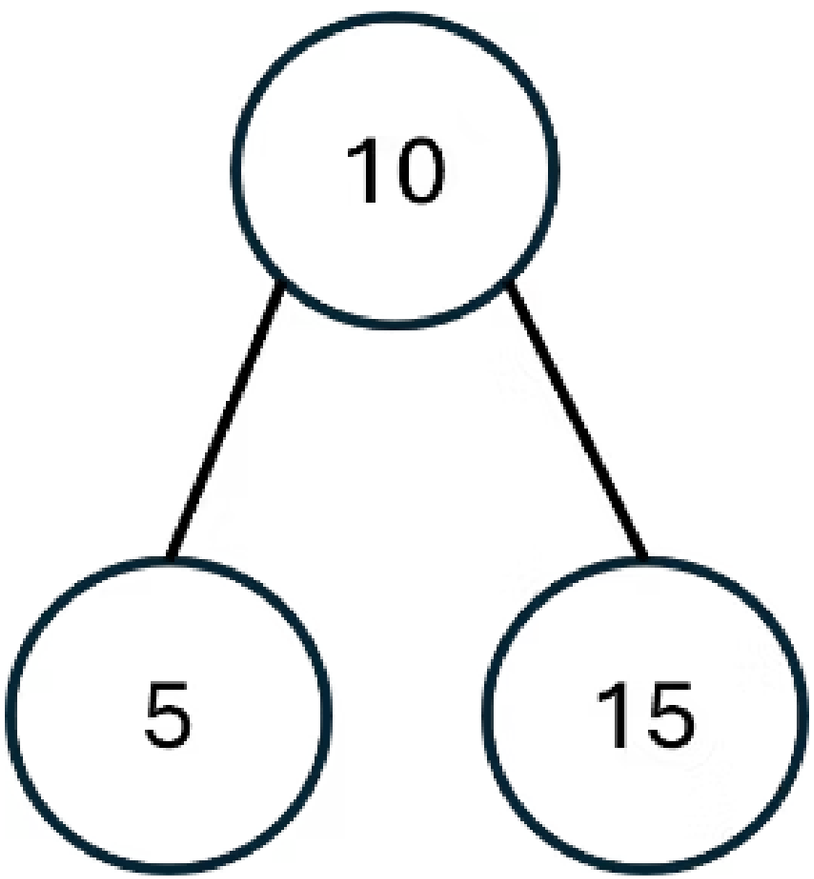
Tipos de Árboles

3.1 Árbol Binario

Descripción: Cada nodo tiene, como máximo, dos hijos denominados hijo izquierdo e hijo derecho.

Aplicación: Se utiliza en algoritmos de recorrido (preorden, inorden, postorden) y como base para estructuras más complejas.

Ejemplo:

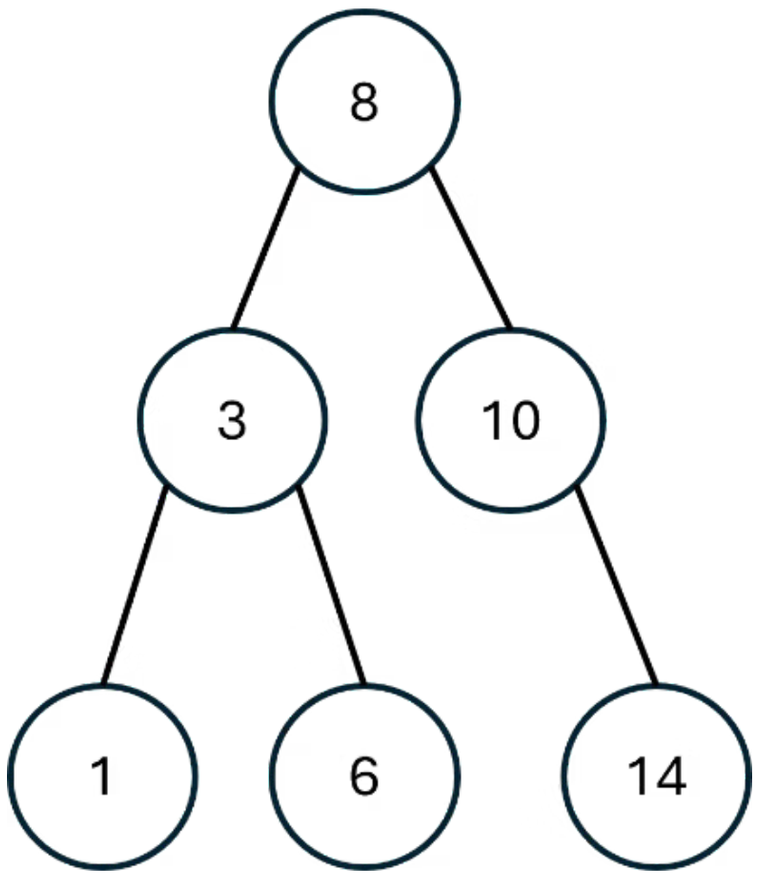


3.2 Árbol de Búsqueda Binaria (ABB)

Descripción: Es un árbol binario donde para cada nodo, todos los elementos del subárbol izquierdo son menores y los del subárbol derecho son mayores que el valor del nodo.

Aplicación: Permite realizar búsquedas, inserciones y eliminaciones de manera eficiente, generalmente en tiempo logarítmico.

Ejemplo:



Más Tipos de Árboles

3.3 Árboles Balanceados

Estas estructuras garantizan que la altura del árbol se mantenga en un nivel óptimo para evitar que las operaciones se degraden a tiempo lineal.

- AVL (Adelson-Velsky y Landis): Se auto-balancea después de cada inserción o eliminación, asegurando que la diferencia de alturas entre subárboles de cualquier nodo sea de, a lo sumo, 1.
- Árbol Rojo-Negro (Red-Black Tree): Cada nodo posee un color (rojo o negro) y se mantienen ciertas propiedades para asegurar un equilibrio razonable. Es menos restrictivo que un AVL, lo que puede resultar en menos rotaciones durante las operaciones, a costa de un árbol menos "estricto" en su balance.

3.4 Árboles B y B+

- Árbol B: Es una estructura de árbol de búsqueda que permite más de dos hijos por nodo, optimizada para sistemas de almacenamiento en disco. Se utiliza ampliamente en bases de datos y sistemas de archivos para gestionar grandes volúmenes de datos.
- Árbol B+: Es una variación del árbol B en el que los datos se almacenan solo en las hojas y estas se encuentran enlazadas entre sí, lo que facilita la realización de búsquedas secuenciales.

3.5 Árboles N-arios

Cada nodo puede tener hasta N hijos, sin la restricción de dos hijos como en el árbol binario.

Aplicación: Son útiles en contextos donde cada nodo debe representar múltiples opciones, como en la representación de directorios de un sistema operativo o en el modelado de juegos de estrategia.

Implementación de Árboles en Python

Se puede representar un nodo como una clase con atributos para el valor y referencias a los hijos.

Ejemplo simple: un árbol binario (cada nodo tiene máximo dos hijos).

Ejemplo de clase Nodo para un árbol binario

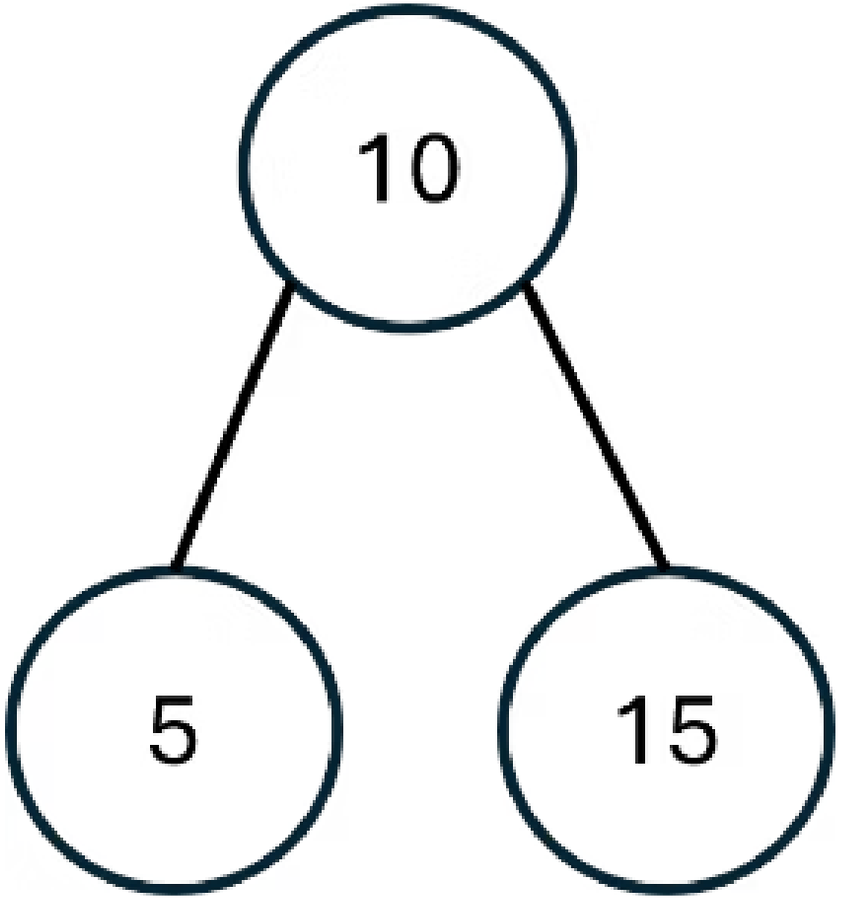
```python
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierdo = None
        self.derecho = None

# Creación de un árbol binario simple
raiz = Nodo(10)
raiz.izquierdo = Nodo(5)
raiz.derecho = Nodo(15)

# Acceso a los valores
print(raiz.valor)            # 10
print(raiz.izquierdo.valor)  # 5
print(raiz.derecho.valor)    # 15
```

```
10
5
15
```

Representación Gráfica



Estos distintos tipos de árboles permiten abordar problemas complejos optimizando la búsqueda, inserción y eliminación de datos. La elección del árbol adecuado depende de las necesidades específicas de la aplicación, tales como la velocidad de las operaciones, la cantidad de datos y el entorno de ejecución (por ejemplo, memoria principal o almacenamiento en disco).Implementar y comprender estas estructuras en Python no solo mejora la eficiencia del código, sino que también sienta las bases para el desarrollo de algoritmos avanzados en machine learning y procesamiento de datos.

4. GRAFOS (GRAPHS)

Un grafo es una estructura de datos compuesta por nodos (o vértices) y aristas (o enlaces) que conectan dichos nodos. Se utiliza para modelar relaciones entre entidades (redes sociales, rutas, conexiones, etc.).

Conceptos Básicos

- Vértices (nodos): entidades del grafo.
- Aristas: conexiones entre nodos (pueden ser dirigidas o no dirigidas).
- Pesos: en grafos ponderados, cada arista puede tener un valor numérico.

En la teoría de grafos existen varias clasificaciones que ayudan a identificar y utilizar las características específicas de cada tipo de grafo según el problema a resolver. A continuación, se describen los principales tipos de grafos:

Tipos de Grafos

4.1 Grafos Dirigidos vs. No Dirigidos

- Grafos Dirigidos (Digraphs): En un grafo dirigido, cada arista tiene una dirección, lo que significa que la conexión entre dos nodos se establece en un sentido específico. Por ejemplo, si existe una arista de A a B, no implica necesariamente que exista una arista de B a A. Este tipo de grafos se utilizan en modelados donde el orden o la dirección es importante, como en redes de flujo, relaciones de precedencia, o modelos de redes sociales (por ejemplo, Twitter, donde un usuario puede seguir a otro sin reciprocidad).
- Grafos No Dirigidos: En los grafos no dirigidos, las aristas no tienen una dirección específica. Si existe una conexión entre dos nodos, se considera bidireccional: A está conectado con B y viceversa. Se usan cuando la relación entre nodos es simétrica, como en redes sociales de amistad o en mapas de carreteras.

4.2 Grafos Ponderados vs. No Ponderados

- Grafos Ponderados: Cada arista en un grafo ponderado tiene asignado un peso o costo, que puede representar distancia, tiempo, capacidad o cualquier otra medida cuantitativa. Estos grafos son muy útiles en problemas de optimización, como encontrar la ruta más corta en un mapa (por ejemplo, el algoritmo de Dijkstra).
- Grafos No Ponderados: En estos grafos, las aristas no tienen un peso asociado. La única información es la conexión entre los nodos. Se utilizan en situaciones donde el "costo" de cada conexión es igual o no relevante, como en algunos algoritmos de búsqueda (por ejemplo, BFS).

Más Tipos de Grafos y su Implementación

4.3 Grafos Conexos vs. No Conexos

- Grafos Conexos: En un grafo conexo, existe al menos un camino entre cualquier par de nodos. Esto implica que el grafo es "una sola pieza" y se pueden alcanzar todos los nodos desde cualquier nodo de partida.
- Grafos No Conexos: En estos grafos, hay al menos un par de nodos entre los cuales no existe ningún camino. Es posible que el grafo esté compuesto por varias "componentes conexas" independientes.

4.4 Grafos Cíclicos vs. Acíclicos

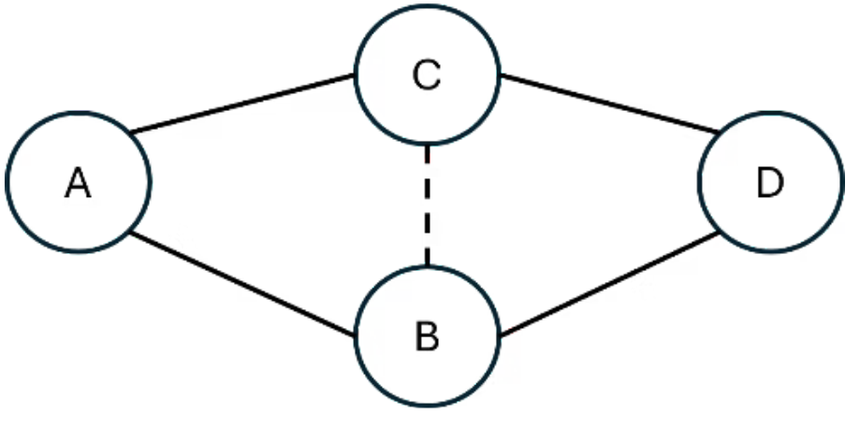
- Grafos Cíclicos: Un grafo cíclico es aquel que contiene al menos un ciclo, es decir, una secuencia de aristas y nodos donde se puede volver al nodo inicial sin repetir una arista.
- Grafos Acíclicos: Un grafo acíclico no contiene ciclos. En el caso de grafos dirigidos, se denominan DAG (Directed Acyclic Graphs) y se utilizan en aplicaciones como la planificación de tareas, ordenamientos topológicos y sistemas de dependencias.

Implementación en Python

Se puede usar un diccionario donde las claves son los nodos y los valores son listas (o diccionarios) de adyacencia.

Ejemplo de grafo simple usando un diccionario:

Representación Gráfica:



A continuación, se presenta un ejemplo simple de cómo representar un grafo no dirigido y no ponderado:

```
# Representación de un grafo no dirigido y no ponderado usando un diccionario
grafo = {
    "A": ["B", "C"],
    "B": ["A", "D", "E"],
    "C": ["A", "F"],
    "D": ["B"],
    "E": ["B", "F"],
    "F": ["C", "E"]
}

print(grafo)
```

```
{'A': ['B', 'C'], 'B': ['A', 'D', 'E'], 'C': ['A', 'F'], 'D': ['B'], 'E': ['B', 'F'], 'F': ['C', 'E']}
```

En este ejemplo, la lista de adyacencia para cada nodo muestra a qué otros nodos está conectado, sin implicar dirección ni pesos en las conexiones.

Esta clasificación de grafos es fundamental para entender cuál es la estructura más adecuada al modelar distintos problemas, ya que cada tipo ofrece ventajas y desventajas específicas para la resolución de problemas complejos. La correcta elección del tipo de grafo permite diseñar algoritmos más eficientes y ajustados a las necesidades del problema en cuestión.

5. Casos y Ejemplos de Uso Avanzado

Uso de Estructuras de Datos en Aprendizaje de Máquina

- Listas y tuplas para almacenar vectores de características o configuraciones de hiperparámetros.
- Diccionarios para mapear nombres de modelos a configuraciones o resultados.
- Sets para filtrar datos duplicados o para verificar pertenencia de valores.
- Pilas y colas en la implementación de búsquedas o algoritmos que requieran LIFO/FIFO (p. ej. backtracking, BFS, DFS).
- Árboles para representar árboles de decisión, jerarquías de clasificación o modelos basados en árboles (p. ej. Random Forest).
- Grafos para problemas de redes neuronales (conceptualmente), redes de transporte o análisis de redes sociales.

6. Actividad Práctica Guiada

Objetivo: Implementar un pequeño sistema de manejo de datos de una red social básica, donde se utilicen diccionarios para almacenar la información de los usuarios y un grafo para representar sus conexiones.

Instrucciones Paso a Paso:

1. Crear un diccionario para la información de cada usuario
    - Cada clave será el username del usuario.
    - Cada valor será otro diccionario con campos como nombre, edad y amigos (lista de usernames amigos).
2. Representar la red de amistades como un grafo
    - Crear un diccionario red_amistades donde cada clave es un username y cada valor es la lista de amigos.
3. Agregar usuarios
    - Implementar una función agregar_usuario(username, nombre, edad) que actualice el diccionario de usuarios.
4. Conectar usuarios
    - Implementar una función agregar_amigo(user1, user2) que actualice la lista de amigos en red_amistades para ambos usuarios (grafo no dirigido).
5. Visualizar información
    - Mostrar el diccionario de usuarios y el grafo de amistades.

Fragmento de Código (como imagen) - Ejemplo

```python
usuarios = {}
red_amistades = {}

def agregar_usuario(username, nombre, edad):
    usuarios[username] = {
        "nombre": nombre,
        "edad": edad,
        "amigos": []
    }
    red_amistades[username] = []

def agregar_amigo(user1, user2):
    if user2 not in red_amistades[user1]:
        red_amistades[user1].append(user2)

    if user1 not in red_amistades[user2]:
        red_amistades[user2].append(user1)

    usuarios[user1]["amigos"].append(user2)
    usuarios[user2]["amigos"].append(user1)

# Ejemplo de uso
agregar_usuario("alice", "Alice", 30)
agregar_usuario("bob", "Bob", 25)
agregar_amigo("alice", "bob")

print(usuarios)
print(red_amistades)
```

```
{'alice': {'nombre': 'Alice', 'edad': 30, 'amigos': ['bob']}, 'bob': {'nombre': 'Bob', 'edad': 25, 'amigos': ['alice']}}
{'alice': ['bob'], 'bob': ['alice']}
```

Resultado Esperado:

- Verás en usuarios un diccionario con la información de alice y bob, cada uno con la lista de amigos correspondiente.
- En red_amistades, observarás un diccionario que representa el grafo de conexiones.