TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

TRABAJO INTEGRADOR PROGRAMACIÓN I

Estructuras de datos avanzadas: árboles y su implementación en un caso práctico.

Alumnos: ***Montero Morinigo, Ramón Armando***
COMISIÓN 25
rama.montero96@hotmail.com

Capelli Camilo
COMISIÓN 11
camilocapelli@gmail.com

Materia: Programación I
Profesor/a: Sebastián Bruselario
Tutor: Verónica Carbonari
Fecha de Entrega: 09/06/2025

ÍNDICE

---

1. Introducción

Este trabajo tiene como finalidad centrarse en el estudio y la aplicación práctica de estructuras de datos avanzadas, en particular, los árboles. Los árboles son una de las estructuras más fundamentales y poderosas en informática: permiten organizar datos de forma jerárquica y eficiente, facilitando operaciones como búsquedas, inserciones y eliminaciones.

El objetivo principal del proyecto es desarrollar un programa que simule la creación de un fixture deportivo. A partir del ingreso manual de los equipos participantes, el sistema genera un esquema de eliminación directa, simula los partidos y determina un campeón.

La relevancia de este proyecto dentro del área de la programación radica en que integra el uso de clases, estructuras de árbol, lógica condicional, recursividad y simulación, todo en un contexto práctico y cercano a una situación real. Además, representa una oportunidad para afianzar el uso de estructuras jerárquicas y el razonamiento en Python.

---

2. Marco teórico

Los árboles son estructuras de datos no lineales ampliamente utilizadas en informática para representar relaciones jerárquicas entre elementos. A diferencia de listas o arreglos, que almacenan datos de manera secuencial, los árboles permiten organizar la información en distintos niveles, facilitando operaciones como búsqueda, inserción, eliminación y recorrido de datos con gran eficiencia.

Un árbol está compuesto por nodos conectados entre sí a través de ramas. Cada nodo puede tener un valor y referencias a otros nodos denominados hijos. El nodo ubicado en la parte superior se llama raíz y a partir de él se extiende toda la estructura. Los nodos sin hijos se denominan hojas, mientras que aquellos que tienen al menos un hijo se consideran nodos internos. Además, existen relaciones familiares entre los nodos: por ejemplo, cada nodo (excepto la raíz) tiene un padre, y los nodos que comparten el mismo padre son hermanos.

Desde una perspectiva más formal, un árbol puede visualizarse como un tipo especial de grafo acíclico dirigido, donde no existen ciclos y siempre se parte desde una raíz única. Algunas de las propiedades importantes son la profundidad de un nodo, el nivel y la altura del árbol. También se habla de grado de un nodo (cantidad de hijos que tiene) y peso del árbol (cantidad total de nodos).

Un subtipo fundamental de árbol es el árbol binario, en el que cada nodo tiene como máximo dos hijos, comúnmente denominados hijo izquierdo e hijo derecho. Cuando estos árboles respetan ciertas reglas de orden (por ejemplo, los menores a la izquierda y los mayores a la derecha), se denominan árboles binarios de búsqueda, ampliamente usados en algoritmos eficientes de búsqueda y ordenamiento.

Existen diversas maneras de representar y recorrer árboles. Entre las representaciones gráficas se destacan el uso de grafos, listas anidadas o estructuras con indentación. En cuanto a los recorridos, los más comunes son preorden, inorden y postorden, cada uno útil para distintos propósitos como copiar estructuras, recorrer en orden o realizar operaciones jerárquicas desde las hojas hacia la raíz.

Comprender los árboles es esencial en programación, ya que aparecen en múltiples contextos: estructuras de archivos, bases de datos jerárquicas, árboles genealógicos, evaluadores de expresiones matemáticas y sistemas de decisión, entre otros. Además, su implementación en lenguajes de alto nivel como Python permite aplicar conceptos como recursividad y manejo de objetos, lo que los convierte en un recurso clave para resolver problemas complejos de forma eficiente.

---

3. Caso práctico

El presente desarrollo tiene como objetivo aplicar conceptos fundamentales de estructuras de datos, árboles binarios y simulación de un caso práctico: la generación y simulación de un fixture de eliminación directa para un torneo de fútbol. A través del lenguaje de programación Python, se buscó modelar el recorrido completo de un torneo desde su fase inicial hasta la consagración del campeón, utilizando una estructura en forma de árbol binario para representar los partidos y sus resultados.

Desde la teoría, un árbol binario es una estructura jerárquica donde cada nodo tiene como máximo dos hijos, lo cual se adapta perfectamente al formato de torneos "play-off", donde cada partido genera un solo ganador que avanza de ronda. Esta estructura permite representar gráficamente y computacionalmente el flujo de un torneo donde los equipos se enfrentan en rondas sucesivas (octavos, cuartos, semifinales, etc.) hasta llegar a la final.

Además, se integran conceptos como la recursividad para recorrer el árbol y simular los resultados de cada encuentro, y estructuras auxiliares como deque* para organizar e imprimir los partidos por rondas, desde la inicial hasta la final. De esta manera, se pone en práctica la teoría vista en algoritmos y estructuras de datos, resolviendo un problema cotidiano del ámbito deportivo mediante herramientas de programación y lógica computacional.

***Colas Dobles (Deque):** La clase deque del módulo collections en Python proporciona una cola doblemente terminada, lo que significa que se pueden agregar y eliminar elementos de ambos extremos de manera

eficiente. Es útil para recorrer niveles de un árbol de manera ordenada. En este caso, utilizamos deque para almacenar y procesar los nodos del árbol de partidos por niveles.

Desarrollo del caso: Implementación en la práctica

1. Definición de la Clase Partido: Cada partido es representado como un nodo con dos equipos enfrentándose. La clase Partido tiene atributos para los equipos, el ganador y los nodos izquierdo y derecho que representan las rondas anteriores.

```python
# Clase Partido (Nodo)
class Partido:
    def __init__(self, equipo1=None, equipo2=None):
        self.equipo1 = equipo1
        self.equipo2 = equipo2
        self.ganador = None
        self.izquierdo = None
        self.derecho = None

    def simular(self):
        if self.equipo1 and self.equipo2:
            self.ganador = random.choice([self.equipo1, self.equipo2])
        elif self.equipo1:
            self.ganador = self.equipo1
        elif self.equipo2:
            self.ganador = self.equipo2
        else:
            self.ganador = None
        return self.ganador
```

2. Construcción del Árbol de Partidos: Se construye un árbol binario donde cada nivel representa una ronda del torneo. Se emparejan los equipos y se crean nodos Partido para cada enfrentamiento. Luego, se agrupan los nodos en rondas sucesivas hasta formar el árbol

completo.

```python
# Construcción del árbol
def construir_fixture(equipos):
    nodos = [Partido(e1, e2) for e1, e2 in zip(equipos[::2], equipos[1::2])]

    while len(nodos) > 1:
        siguiente_ronda = []
        for i in range(0, len(nodos), 2):
            padre = Partido()
            padre.izquierdo = nodos[i]
            padre.derecho = nodos[i+1] if i+1 < len(nodos) else None
            siguiente_ronda.append(padre)
        nodos = siguiente_ronda
    return nodos[0]
```

3. Simulación del Torneo: Se simula el torneo recorriendo el árbol desde las hojas (partidos iniciales) hasta la raíz (final), determinando los ganadores de cada partido. Se utiliza recursión para recorrer el árbol y simular cada partido, asignando el ganador correspondiente.

```python
# Simulación del torneo
def simular_torneo(nodo):
    if nodo is None:
        return None
    if nodo.izquierdo is None and nodo.derecho is None:
        nodo.simular()
        return nodo.ganador
    nodo.equipo1 = simular_torneo(nodo.izquierdo)
    nodo.equipo2 = simular_torneo(nodo.derecho)
    nodo.simular()
    return nodo.ganador
```

4. Validar la cantidad de equipos: Antes de construir el fixture, se valida que el número de equipos ingresado por el usuario sea una potencia de 2 (2, 4, 8, 16), ya que un torneo de eliminación directa requiere que todos los equipos puedan enfrentarse sin quedar equipos sin rival en ninguna ronda. Se usa una operación binaria eficiente que verifica si n es una potencia de 2. Esto se aplica luego al momento de ingresar la cantidad de equipos:

```python
# Validar cantidad de equipos
def es_potencia_de_dos(n):
    return n >= 2 and (n & (n - 1)) == 0
```

5. Definir nombres de etapas según la cantidad de equipos (profundidad): Se utiliza un diccionario que asocia niveles del árbol (profundidad) con el nombre de la fase del torneo. Esto permite que, al momento de mostrar los resultados del fixture, cada partido esté

acompañado por el nombre de su ronda correspondiente, mejorando la legibilidad del simulador.

```python
# Definir nombres de etapas según la cantidad de equipos (profundidad)
etapas_nombre = {
    1: "Final",
    2: "Semifinales",
    3: "Cuartos de final",
    4: "Octavos de final",
    5: "Dieciseisavos de final"
}
```

6. Calcular profundidad máxima del árbol: Para poder determinar en qué ronda se encuentra cada partido, primero hay que saber cuál es la profundidad máxima del árbol (es decir, cuántos niveles tiene desde los partidos iniciales hasta la final). Este algoritmo recorre el árbol de manera recursiva, obteniendo la profundidad del subárbol izquierdo y del derecho, y devuelve el máximo.

```python
# Calcular profundidad máxima del árbol
def profundidad_arbol(nodo):
    if nodo is None:
        return 0
    izq = profundidad_arbol(nodo.izquierdo)
    der = profundidad_arbol(nodo.derecho)
    return max(izq, der) + 1
```

7. Mostrar el fixture por niveles: Se recorre el árbol por niveles utilizando una cola doble (deque) para almacenar los nodos de cada nivel. Se implementa un recorrido por niveles del

árbol para mostrar los partidos de cada ronda en orden.

```python
# Mostrar fixture nivel por nivel desde ronda inicial hasta final
def mostrar_fixture_por_niveles(raiz):
    if raiz is None:
        return

    profundidad_max = profundidad_arbol(raiz)
    cola = deque()
    cola.append((raiz, 1))  # nodo y nivel

    niveles = {}

    while cola:
        nodo, nivel = cola.popleft()
        if nivel not in niveles:
            niveles[nivel] = []
        niveles[nivel].append(nodo)

        if nodo.izquierdo:
            cola.append((nodo.izquierdo, nivel + 1))
        if nodo.derecho:
            cola.append((nodo.derecho, nivel + 1))

    # Imprimir desde nivel máximo (ronda inicial) hasta 1 (final)
    for nivel in range(profundidad_max, 0, -1):
        etapa = etapas_nombre.get(nivel, f"Ronda {nivel}")
        print(f"\n[{etapa}]")
        for partido in niveles.get(nivel, []):
            if partido.equipo1 and partido.equipo2:
                print(f"  {partido.equipo1} vs {partido.equipo2} → {partido.ganador}")
```

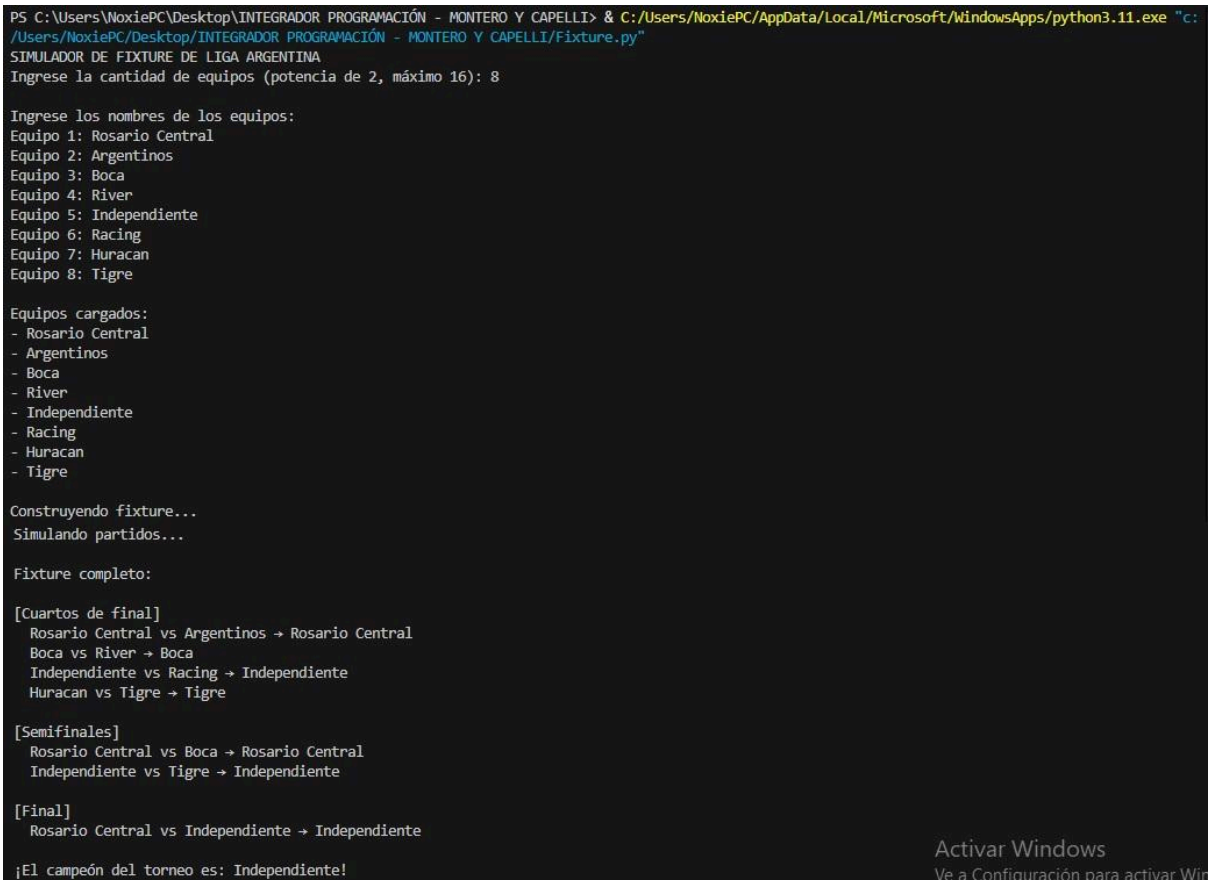
8. Programa principal: La sección principal del programa integra todos los elementos: validación de datos, ingreso de equipos, construcción del árbol, simulación del torneo y visualización del fixture completo con rondas etiquetadas. Esta sección coordina todo el proceso. Una vez validados los equipos y construida la estructura del torneo, se simulan los partidos y se

imprime un resumen por etapas, incluyendo el campeón.

```python
# Programa Principal
if __name__ == "__main__":
    print("SIMULADOR DE FIXTURE DE LIGA ARGENTINA")

    while True:
        try:
            cantidad = int(input("Ingrese la cantidad de equipos (potencia de 2, máximo 16): "))
            if not es_potencia_de_dos(cantidad):
                print("La cantidad debe ser una potencia de 2.")
            elif cantidad > 16:
                print("La cantidad máxima permitida es 16.")
            else:
                break
        except ValueError:
            print("Ingrese un número válido.")

    equipos = []
    print("\nIngrese los nombres de los equipos:")
    for i in range(cantidad):
        nombre = input(f"Equipo {i+1}: ")
        equipos.append(nombre.strip())

    print("\nEquipos cargados:")
    for e in equipos:
        print(f"- {e}")

    print("\nConstruyendo fixture...")
    raiz = construir_fixture(equipos)

    print("\nSimulando partidos...")
    campeon = simular_torneo(raiz)

    print("\nFixture completo:")
    mostrar_fixture_por_niveles(raiz)

    print(f"\n¡El campeón del torneo es: {campeon}!")
```

9. Ejecución del programa: Al ejecutar la consola nos pide ingresar el número de equipos que participarán del torneo. Luego pasará a pedirnos el ingreso de esos equipos. Y una vez verificados estos datos procederá a simular el torneo con sus respectivas instancias

eliminatorias hasta concluir con el campeón del mismo.



La implementación del simulador de torneo se basó en principios fundamentales de estructuras de datos, como los árboles binarios y las colas dobles, para modelar y simular eficientemente un torneo eliminatorio. La teoría se aplicó de manera práctica para construir una solución funcional y estructurada.

---

4. Metodología utilizada

El desarrollo del trabajo se estructuró en distintas etapas, priorizando una comprensión práctica de estructuras de datos avanzadas, en particular, los árboles:

Investigación inicial:

Se consultó el material teórico y audiovisual brindado en el campus virtual, junto con documentación oficial y bibliografía complementaria. Esta etapa permitió comprender los conceptos fundamentales relacionados con los árboles binarios, su aplicación en programación, su implementación en Python

y, de esa forma, poder definir una forma adecuada y cercana a una situación real para la explicación práctica.

Planificación y pruebas:

Para resolver el problema de simular un fixture de torneo en formato de eliminación directa, se diseñó un modelo basado en un árbol, donde cada nodo representa un partido y sus nodos hijos corresponden a los partidos previos de los que emergen los equipos ganadores.

Herramientas y recursos utilizados:

- Python como lenguaje para el desarrollo del programa.
- Visual Studio Code como entorno de desarrollo.
- GitHub como plataforma de control de versiones.
- Documentos de texto y recursos audiovisuales de TUP.

Trabajo colaborativo:

Para facilitar el trabajo en equipo y asegurar un desarrollo ordenado del proyecto, se utilizó GitHub como plataforma de control de versiones. Se creó un repositorio público donde ambos integrantes del grupo tuvieron acceso para editar, subir cambios y revisar el código de Python de forma colaborativa. El desarrollo del programa se realizó utilizando Python, en Visual Studio Code y la estructura del código fue escrita respetando indentación, uso de comentarios explicativos, nombres de variables significativos y separación entre secciones.

---

5. Resultados obtenidos

- El programa funciona correctamente hasta 16 equipos.
- Se generó un fixture completo.
- Se logró simular la competencia y determinar un campeón.
- Se validó el ingreso correcto de datos por parte del usuario.
- La lógica de árbol permitió manejar la estructura del torneo sin complicaciones.

---

6. Conclusiones

Este trabajo permitió comprender la importancia de los árboles como estructura útil en la programación práctica. Aplicarlos a un contexto deportivo facilitó su comprensión y permitió un desarrollo funcional y realista.

---

7. Bibliografía

- Python Software Foundation. (2024). Python 3 Documentation. https://docs.python.org/3/
- Python Software Foundation. (2024). *collections — Container datatypes*. Python 3.11.7 documentation. https://docs.python.org/3/library/collections.html#collections.deque
- Schmidt, R. (n.d.). *collections.deque — Double-ended queue*. Python Module of the Week. https://rico-schmidt.name/pymotw-3/collections/deque.html
- YouTube - Arboles parte 1, 2 3 y 4 - https://www.youtube.com/watch?v=cVm51pO35qA&list=PLy5wpwhsM-2lIY-qe_fALJ4K_XAhLZ2l-&index=4 https://www.youtube.com/watch?v=kqmn7AYYdSA https://www.youtube.com/watch?v=I4IKGp0PsO0&list=PLy5wpwhsM-2lIY-qe_fALJ4K_XAhLZ2l- https://www.youtube.com/watch?v=O5gvB-LWyhY&list=PLy5wpwhsM-2lIY-qe_fALJ4K_XAhLZ2l-&index=2
- YouTube - Implementación de datos como listas anidadas - https://www.youtube.com/watch?v=-D4SxeHQGIg
- Programación I. (2025). *Documento de texto teórico titulado "Árboles"*. Plataforma institucional de TUP.

---

8. Anexos

- Capturas del programa en ejecución.
- Enlace al repositorio: https://github.com/RamaMontero/TrabajoIntegradorProg/tree/main
- Enlace al video tutorial: https://www.youtube.com/watch?v=RwT11zFu4E4
- Código completo en archivo (py.).