Mini Proyecto ADA-1

Julián Ernesto Puyo Mora *2226905* Laura Camila Betancourt Horta *2223435*
Jhoan Felipe León Correa *2228527* Juan Camilo Narváez Tascón *2140112*

Junio 16, 2024

Materia: Análisis y Diseño de Algoritmos 1
Universidad: Universidad del Valle
Repositorio: https://github.com/Ulvenforst/MiniProyectoADA1/tree/main

Contents

1 Introducción

En el presente informe se expone el desarrollo y análisis de dos soluciones para organizar un sistema de jugadores, equipos, sedes y asociaciones deportivas, empleando el paradigma de programación orientada a objetos en Python. Este proyecto se enmarca en el curso de Análisis y Diseño de Algoritmos I y se centra en el diseño e implementación de algoritmos de ordenamiento eficientes para manejar grandes conjuntos de datos relacionados con el rendimiento deportivo.

La organización o `Asociacion` tiene K `sedes` por todo el país, cada `Sede` un `nombre` y M `equipos` para diferentes deportes, a su vez `Equipo` tiene un `deporte` definido y está formado por una cantidad mínima $N_{\min}$ y máxima $N_{\max}$ de `jugadores`. Cada `Jugador` tendrá un número como `identificador`, su `nombre`, `edad` y `rendimiento`, este último tomará valores $i \in N : 1 \leq i \leq 100$.

La `Asociacion` busca que los `equipos` internamente estén ordenados ascendentemente teniendo en la cuenta el rendimiento de los `jugadores`, en caso de empate se colocará primero el `Jugador` de mayor edad. También para cada `Sede`, se busca que los `equipos` estén ordenados ascendentemente por su rendimiento promedio, el cual se define como la suma de los rendimientos de los `jugadores` del equipo sobre la cantidad de `jugadores` del equipo:

$$\text{rendimiento\_promedio} = \frac{\sum_{i=1}^{N} \texttt{Jugador}_i\texttt{.rendimiento}}{N}$$

En caso de empate en el valor del rendimiento entre `equipos`, **se pondrá primero el** `Equipo` **que tiene mayor cantidad de** `jugadores`. Para las `sedes` se busca algo similar, se requiere ordenar las `sedes` de forma ascendente según el promedio de los rendimientos de todos los equipos de la respectiva `Sede`, en caso de que dos `sedes` tengan el mismo rendimiento promedio la `Sede` con más `jugadores` se pondrá primero. Por último con el fin de saber el ranking de los jugadores para tomar decisiones respecto a esto, se requiere también generar la lista de todos los `jugadores` de todas las `sedes` **ordenados por su rendimiento ascendentemente**.

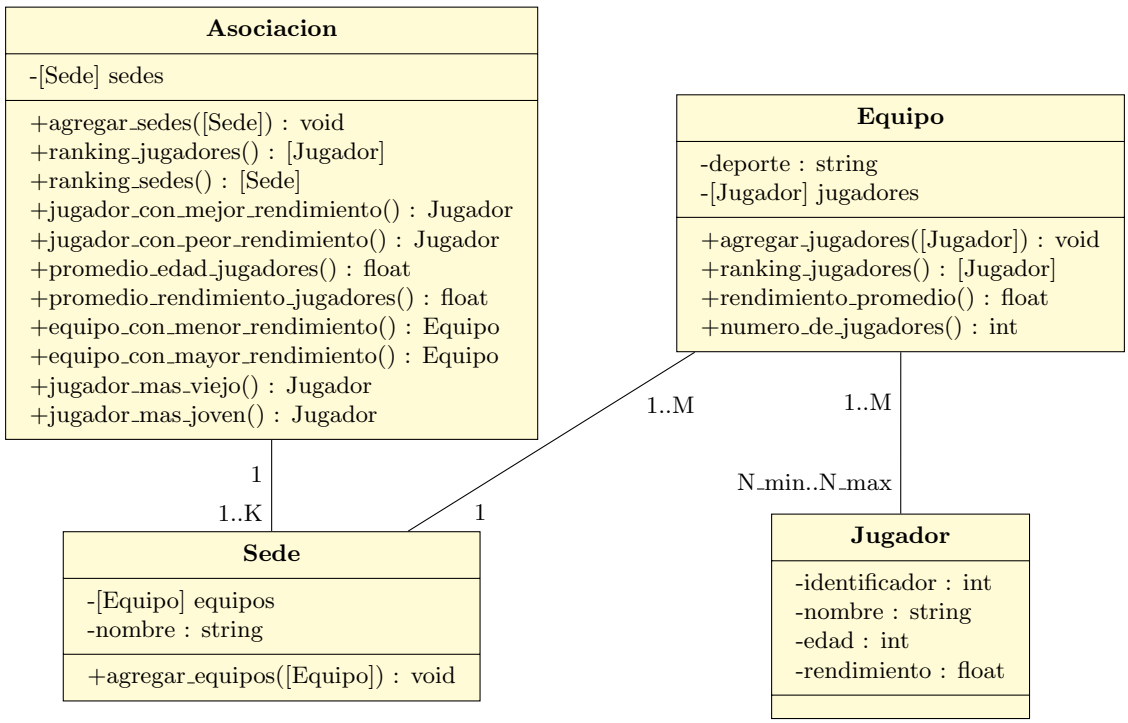
La necesidad de clasificar estos elementos de acuerdo con diversos criterios, como el rendimiento y la edad, ha llevado a la implementación de dos enfoques distintos: uno basado en **listas**, se han utilizado los algoritmos *Counting Sort* y *Bucket Sort* para el ordenamiento de los datos. Por otro lado, en la solución basada en **árboles rojinegros**, se ha aprovechado el ordenamiento inherente de esta estructura de datos.

El objetivo de este informe es proporcionar un resumen detallado de las soluciones desarrolladas, incluyendo la descripción de los algoritmos de ordenamiento utilizados, su análisis de complejidad y un análisis comparativo de su rendimiento tanto en teoría como en la práctica.

2 Planteamiento del modelo

`src/Listas/classes || src/ArbolRojiNegro/classes`

Tal como fue descrito, se planteó un modelo en Python, basado en el paradigma orientado a objetos, con el fin de mantener la modularidad y el fácil llamado a los diferentes atributos establecidos para las entidades (o clases) `Jugador`, `Equipo`, `Sede` y `Asociación`; tal como se muestra a continuación.



Note que la estructura de datos compuesta puede variar según la solución a emplear, de igual manera según lo necesitado para brindar una solución pueden haber más atributos, sin embargo este diagrama representa de manera general la composición del programa.

3 Descripción de los Algoritmos Basados en Listas (*Arrays*)

`src/Listas`

La idea es representar cada conjunto de clases con una lista, esta la ordenaremos con *counting sort* y *bucket sort* basándonos en los criterios descritos en la introducción. Consecuentemente, solo debemos acceder al primer elemento o último según se necesite. Los atributos auxiliares como promedios o sumas se irán acumulando a medida que se agregan elementos.

En un inicio se planeó el uso de tablas hash como una de las estructuras de datos a emplear, sin embargo, a medida que se avanzó en el desarrollo de la solución, pudimos encontrar ineficiencias; las tablas hash suelen ser una herramienta adecuada para la consulta de valores según una llave, pero en el contexto de nuestro problema específico, donde las operaciones de ordenamiento y rangos son frecuentes y críticas, este tipo de estructura no resultó ser la más eficiente. Aunque las tablas hash proporcionan inserciones y búsquedas en tiempo promedio constante $O(1)$, enfrentan limitaciones cuando se requiere mantener un orden intrínseco entre los elementos, como es necesario para determinar los jugadores con mejor rendimiento o para ordenar equipos y sedes basados en diversas métricas de rendimiento.

Dada la naturaleza del problema decidimos expandir las tablas hash con listas, de manera que trabajaran de manera conjunta, usando las listas donde las tablas hash encontraban limitaciones. Una vez terminamos esta implementación encontramos múltiples redundancias, por ejemplo al ordenar una lista para luego insertar los

datos de manera adecuada en la tabla hash y luego consultar el primer o último elemento de este; lo redundante está en crear una estructura más cuando la lista ya tiene lo que necesitamos. Como se mencionó las tablas hash suelen ser una buena herramienta en la consulta según llaves, pero nuestro problema nos solicitaba encontrar el primer o último elemento de la estructura empleada, y para esto las listas tienen un tiempo constante $O(1)$.

Luego exploramos otras opciones, como las listas enlazadas, aquí nos dimos cuenta de que, aunque las operaciones de inserción y eliminación son teóricamente más rápidas en las listas doblemente enlazadas que en los *arrays*, para nuestro caso específico, donde el acceso rápido y frecuente al primer o último elemento es crítico, los *arrays* resultaron ser más adecuados; los *arrays* permiten acceder a cualquier elemento, incluyendo el primero y el último, en tiempo constante $O(1)$, lo cual es esencial para la eficiencia en la gestión y análisis de rendimiento de equipos y jugadores.

En las operaciones de inserción y eliminación al principio de un *array*, aunque cada elemento debe ser desplazado, lo que teóricamente toma un tiempo lineal $O(n)$, en la práctica, la simplicidad de manejo de memoria y la localidad de referencia ofrecen un rendimiento generalmente más cómodo para trabajar comparado con la gestión de punteros en listas enlazadas (nos referimos a la simulación de punteros en python). Además, note que nuestro problema no pide la eliminación de elementos, y los elementos se ordenan dada una lista ya creada, por lo que solo se debe acceder al k-ésimo elemento una vez ordenado. Esta ventaja es particularmente significativa en nuestro caso específico.

Por otro lado, en contextos donde se requiere el acceso al k-ésimo elemento, los *arrays* destacan sobre las listas enlazadas ya que permiten este acceso de forma directa sin necesidad de recorrer la estructura, manteniendo un rendimiento constante que es crucial para las operaciones de análisis y actualización en tiempo real dentro del sistema de gestión deportiva.

> La *inserción* y *eliminación* son operaciones más rápidas en listas doblemente enlazadas que en arrays. Si se desea insertar un nuevo primer elemento en un array o eliminar el primer elemento de un array, manteniendo el orden relativo de todos los elementos existentes, entonces cada uno de los elementos existentes necesita ser movido una posición. Por lo tanto, en el peor caso, tanto la inserción como la eliminación toman tiempo $\Theta(n)$ en un array, comparado con tiempo $O(1)$ para una lista doblemente enlazada. Sin embargo, si se desea encontrar el k-ésimo elemento en el orden lineal, toma solo $O(1)$ tiempo en un array independientemente de k, pero en una lista enlazada, se tendría que recorrer k elementos, tomando $\Theta(k)$ tiempo.
> —*Introduction to Algorithms, Fourth Edition [Leiserson Stein Rivest Cormen], pg. 283 pdf.*

En resumen, **optamos por utilizar *arrays* debido a su eficiencia en el acceso directo a los elementos** (como lo es el primero o último) y la simplicidad en la manipulación de datos. El balance entre acceso rápido y simplicidad operativa justifica su elección para nuestro proyecto, al igual que las necesidades propias del mismo. Esta decisión se alinea con las necesidades específicas de acceso y gestión eficiente, lo que permite una implementación más directa y un mantenimiento simplificado del sistema. También usamos listas para estudiar la comparación con los árboles rojinegros (la otra solución que empleamos), comparando si realmente hay diferencia entre una estructura de datos usada comúnmente con otra más compleja.

3.1 Algoritmos de ordenamiento para listas

`src/Listas/utils/sorting_algorithms.py`

Dada una lista recibida con diferentes elementos que deben ser ordenados según un criterio numérico, decidimos hacer uso de *counting sort* y *bucket sort* (el cual también hace uso de *insertion sort*). La elección de estos algoritmos se basa en varias características inherentes de los datos con los que trabajamos y los objetivos de rendimiento que buscamos alcanzar. Ambos algoritmos son particularmente adecuados para situaciones donde los datos presentan características que permiten optimizaciones específicas en términos de complejidad temporal y espacial.

- **Counting Sort**, un algoritmo de ordenamiento no comparativo, es altamente efectivo para datos con un rango conocido y limitado, ya que opera contando la ocurrencia de cada valor del conjunto de datos, acumulando un índice de inicio para cada valor y, finalmente, utilizando esa acumulación para posicionar cada elemento directamente en una nueva lista de salida. Esta metodología elimina la necesidad de comparaciones directas entre los elementos, lo que resulta en una eficiencia temporal de $O(n+k)$, donde n es el número de elementos y k es el rango de valores. Dado que la eficacia de *counting sort* depende de que k no sea significativamente mayor que n, es ideal para nuestros datos, donde el rango de valores numéricos es comparativamente bajo y predecible.

Análisis de Complejidad:

- **Tiempo**: $O(n + k)$, donde n es el número de elementos y k es el rango de los valores.
- **Espacio**: $O(k)$, debido al uso de una lista auxiliar para contar las ocurrencias de cada valor.

- **Bucket Sort**, complementa las ventajas de *counting sort* al ser particularmente útil cuando los datos se pueden distribuir de manera uniforme a través de una serie de 'buckets'. Cada 'bucket' se ordena individualmente, típicamente mediante un algoritmo eficiente en casos de pocos datos, como *insertion sort*. La eficacia de *bucket sort* proviene de su capacidad para distribuir los elementos de manera que los 'buckets' sean de tamaño relativamente uniforme y cada uno contenga un subconjeto del rango de datos, lo que permite que la ordenación dentro de cada 'bucket' sea extremadamente rápida. En escenarios donde los datos están bien distribuidos, *bucket sort* puede acercarse a una complejidad temporal de $O(n)$, lo cual es óptimo para listas más grandes.
  Análisis de Complejidad:
  - **Tiempo**: $\Omega(n + k)$, donde n es el número de elementos y k es el número de buckets. En el caso promedio, si los elementos están distribuidos uniformemente, el tiempo puede reducirse a $O(n)$. $O(n^2)$ si todos los elementos terminan en el mismo bucket y se usa un algoritmo de ordenamiento cuadrático (*insertion sort*).
  - **Espacio**: $O(n + k)$, debido a la necesidad de espacio adicional para los buckets.

La combinación de *counting sort* y *bucket sort* se justifica en nuestro contexto debido a la capacidad de manejar grandes volúmenes de datos con eficiencia, minimizando la sobrecarga de las operaciones de comparación y maximizando el uso de la estructura de datos para reducir el tiempo de procesamiento general. Estos métodos permiten una implementación que puede adaptarse dinámicamente a la naturaleza de los datos recibidos, optimizando el rendimiento y proporcionando una solución robusta y escalable para el problema de ordenación numérica. La adaptabilidad y la eficiencia en la gestión de la complejidad espacial también son factores críticos en la elección de estos algoritmos, asegurando que el uso del espacio sea mínimo y directamente relacionado con la naturaleza de los datos procesados.

4 Descripción de los Algoritmos Basados en Árboles Rojinegros

`src/ArbolRojiNegro`

Los árboles rojinegros son un tipo de árbol binario de búsqueda autoequilibrado que cumple con ciertas propiedades para mantener su equilibrio. Estas propiedades aseguran que las operaciones de búsqueda, inserción y eliminación se puedan realizar en tiempo logarítmico. Las propiedades específicas de un árbol rojinegro son las siguientes:

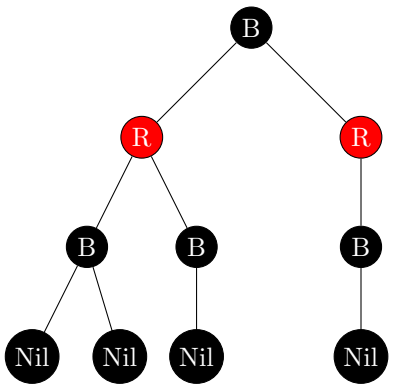


Figure 1: Ejemplo de un árbol rojinegro

1. **Nodo rojo o negro**: Cada nodo es rojo o negro.
2. **Raíz negra**: La raíz del árbol es negra.
3. **Hojas negras**: Todas las hojas (null) son negras. A menudo, se representan como nodos NIL.
4. **Nodos rojos**: Si un nodo es rojo, ambos hijos son negros (es decir, no puede haber dos nodos rojos consecutivos en un camino desde un nodo a una hoja, esto también se conoce como la propiedad de no tener dos rojos seguidos).
5. **Caminos negros**: Para cada nodo, todos los caminos simples desde el nodo hasta sus hojas descendientes contienen el mismo número de nodos negros.

Estas propiedades mantienen el árbol equilibrado en términos de su altura, asegurando que ninguna rama sea más del doble de la longitud de cualquier otra rama. Este equilibrio relativo es lo que permite que las operaciones básicas (búsqueda, inserción, eliminación) sean eficientes.

Análisis de Complejidad:

- **Insertar y buscar**: $O(\log(n))$, donde n corresponde a la cantidad de elementos y $\log(n)$ corresponde a la altura máxima del árbol.
- **Máximo y mínimo**: $O(\log(n))$, donde n corresponde a la cantidad de elementos y $\log(n)$ corresponde a la altura máxima del árbol.
- **Generar la lista ordenada**: $O(n)$, donde n corresponde a la cantidad de elementos.

Teniendo en cuenta lo anterior y considerando la naturaleza del problema, los árboles rojinegros resultan ser una estructura de datos idónea para brindar una solución eficiente a la problemática planteada.

La implementación de los árboles rojinegros para la solución del problema consistió en almacenar jugadores (según rendimiento y edad), equipos y sedes en esta estructura de datos, siguiendo las relaciones establecidas entre las clases. De esta manera, cada una de las clases cuenta con un árbol rojinegro que almacena sus datos, facilitando así su acceso y manejo para alcanzar el objetivo final.

4.1 Análisis Comparativo y de Rendimiento

`src/ArbolRojiNegro/data_structures`

Dado que los algoritmos implementados para las operaciones básicas no sufrieron cambios significativos con respecto a los algoritmos presentados para los árboles rojinegros durante el curso, se garantiza la prevalencia de la complejidad computacional anteriormente mencionada, que se encuentra estrechamente relacionada a la altura del árbol.

Para demostrar la longitud máxima (altura) de un árbol rojinegro, utilizaremos las propiedades específicas de estos árboles. En particular, nos interesa la propiedad de que todos los caminos desde cualquier nodo hasta sus hojas descendientes contienen el mismo número de nodos negros.

4.1.1 Altura de un Árbol Rojinegro

Denotemos la altura de un árbol rojinegro por h. Queremos demostrar que la altura de un árbol rojinegro con n nodos es $O(\log(n))$.

- **Altura negra** (bh): La altura negra de un nodo es el número de nodos negros en cualquier camino desde ese nodo hasta una hoja.
- **Altura total** (h): La altura total es el número de nodos en el camino más largo desde la raíz hasta una hoja.

Vamos a demostrar primero que para un árbol rojinegro con bh, la altura h está acotada por $2bh$

1. Cada camino simple desde la raíz hasta cualquier hoja tiene exactamente bh nodos negros.
2. Un nodo rojo puede tener a lo sumo un nodo negro como padre, lo que significa que entre dos nodos negros consecutivos, puede haber como máximo un nodo rojo.

Esto implica que:

$$h \leq 2 \cdot bh$$

Vamos a demostrar que el número de nodos n en un árbol rojinegro está relacionado con la altura negra bh: Consideremos la mínima cantidad de nodos que puede tener un árbol rojinegro con altura negra bh. En el caso más desfavorable, todos los nodos son negros excepto los nodos terminales (hojas NIL).

Un árbol rojinegro con altura negra bh tiene al menos $2^{bh} - 1$ nodos. Esto es porque un árbol binario completo de altura bh tendría 2^{bh} hojas, y en un árbol rojinegro, cada camino desde la raíz hasta las hojas incluye exactamente bh nodos negros.

$$n \geq 2^{bh} - 1$$

Tomando el logaritmo en base 2 de ambos lados:

$$\lg(n+1) \geq bh$$

Relacionando altura total y número de nodos, sabemos que $h \leq 2 \cdot bh$ y que $\log_2(n+1) \geq bh$ Usando estas dos relaciones, tenemos:

$$h \leq 2 \cdot \lg_2(n+1)$$

Por lo tanto:

$$h = O(\log n)$$

4.1.2 Demostración por Inducción: Inserción en un Árbol Rojinegro

Para un árbol rojinegro vacío (que se considera una hoja negra), la inserción de un solo nodo:

- Se inserta el nodo como rojo.
- Si el nodo es la raíz, se recolorea a negro.

Después de este proceso, todas las propiedades del árbol rojinegro se mantienen:

1. El nodo es negro.
2. La raíz es negra.
3. Las hojas (nodos NIL) son negras.
4. No hay nodos rojos consecutivos.
5. Todos los caminos desde la raíz hasta las hojas tienen el mismo número de nodos negros (en este caso, uno).

Asumamos que para un árbol rojinegro de tamaño n, todas las propiedades del árbol rojinegro se mantienen después de cualquier inserción.

Consideremos ahora un árbol rojinegro de tamaño $n+1$ después de insertar un nuevo nodo. La inserción inicial se realiza como en un árbol binario de búsqueda estándar y el nuevo nodo se colorea rojo. Esta inserción puede causar una violación de las propiedades del árbol rojinegro, especialmente las propiedades 2 (la raíz es negra) y 4 (no hay dos nodos rojos consecutivos).

1. **El nodo insertado es la raíz**:
   - Si el nodo insertado es la raíz, se recolorea a negro.
   - Las propiedades 2 y 5 se mantienen.
2. **El padre del nodo insertado es negro**:
   - No se viola ninguna propiedad.
   - Todas las propiedades del árbol rojinegro se mantienen.
3. **El padre del nodo insertado es rojo**:
   - Aquí hay dos subcasos dependiendo del color del tío (hermano del padre del nodo insertado):
     - **Caso 1: El tío es rojo**:

        * Recolorear el padre y el tío a negro y el abuelo a rojo.
        * Si el abuelo es la raíz, se recolorea a negro.
        * Esto mantiene las propiedades 1, 2, 4 y 5. Si la violación se desplaza al abuelo, repetimos el proceso.
    - **Caso 2: El tío es negro**:
        * Si el nuevo nodo es hijo derecho de un padre hijo izquierdo (o hijo izquierdo de un padre hijo derecho), realizamos una rotación izquierda (o derecha) en el padre.
        * Realizamos una rotación derecha (o izquierda) en el abuelo.
        * Intercambiamos los colores del padre y del abuelo.
        * Esto corrige las propiedades 1, 2 y 4, asegurando que no haya dos nodos rojos consecutivos y manteniendo el balance de nodos negros (propiedad 5).

Después de las correcciones necesarias, el árbol rojinegro mantiene todas sus propiedades, completando así el paso inductivo y demostrando que la complejidad computacional prevalecerá.

5 Análisis de Resultados

Se toman como casos de prueba los tres *inputs* dados por el enunciado. Estos generan un archivo `src/resultados.txt` creado con cada ejecución del `src/main.py`. Se pueden observar los resultados de ambas soluciones y compararlas con los archivos *output* dados por el enunciado. Los datos para el análisis de tiempo son los siguientes:

- `input1.txt`:
    - **Número de jugadores**: 12
    - **Número de equipos**: 4
    - **Número de sedes**: 2
    - **Número de Asociaciones**: 1
- `input2.txt`:
    - **Número de jugadores**: 20
    - **Número de equipos**: 6
    - **Número de sedes**: 2
    - **Número de Asociaciones**: 1
- `input3.txt`:
    - **Número de jugadores**: 60
    - **Número de equipos**: 15
    - **Número de sedes**: 3
    - **Número de Asociaciones**: 1

Luego de verificar que los resultados obtenidos (reservados en `requerimientos.txt`) fueran correctos, capturamos los resultados en términos de rendimiento. Se llevaron a cabo 5 ejecuciones las cuales, evidenciaron el siguiente comportamiento.

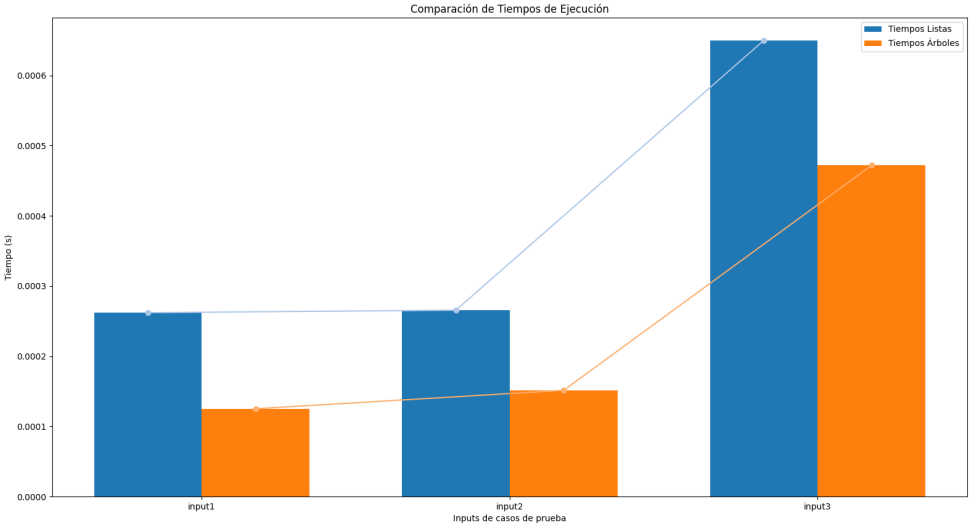


Figure 2: Comparaciones de tamaño de entrada vs tiempo de salida, esto se hace tomando tiempos de ejecución del algoritmo que soluciona el problema. (Ejecución en Linux)

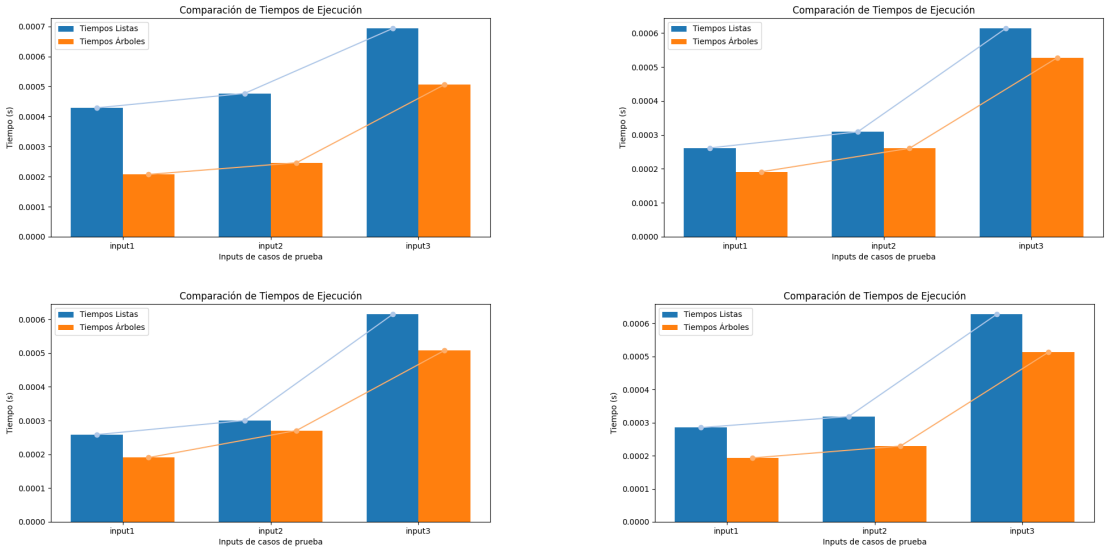


Figure 3: Más instancias de tamaño de entrada vs tiempo de salida para los inputs dados.

Posteriormente se llevaron a cabo una serie de pruebas diseñadas con el fin de explorar el comportamiento de los algoritmos, a través de distintos tamaños de entrada. En nuestro caso usamos los tamaños desde 100 jugadores, hasta 1000 jugadores, incrementando en centenas. Para cada valor de n jugadores se instanciaron $\lfloor n/3 \rfloor$ equipos y $\lfloor n/6 \rfloor$ sedes. Todo bajo el sistema de una sola asociación.

Dichos resultados comparten cierta configuración importante, que vale la pena mencionar, y es que solo se insertan 3 jugadores por equipo y 2 equipos por sede. Lo cual nos dio los siguientes resultados:

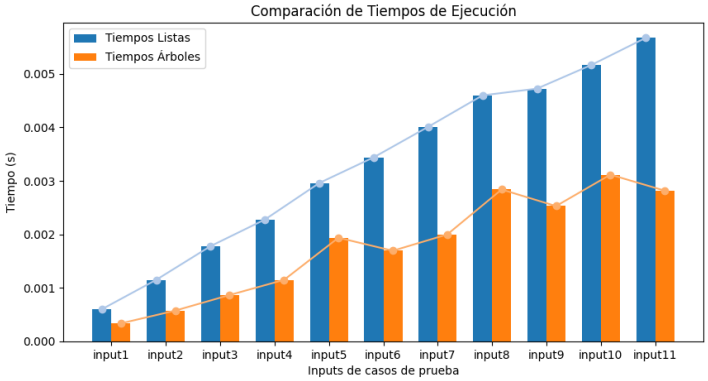


Figure 4: Ejecución 1: Máximo 3 inserciones de jugadores por equipo, y 2 equipos por sede.

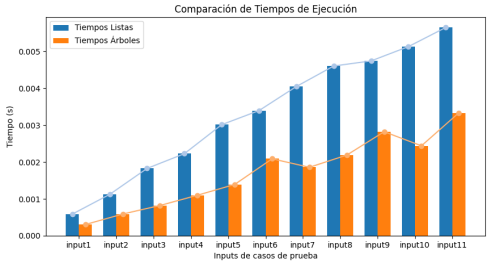


Figure 5: Ejecución 2: Máximo 3 inserciones de jugadores por equipo, y 2 equipos por sede.

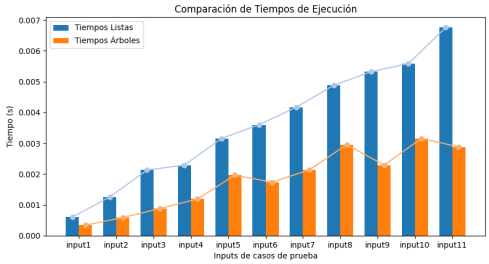


Figure 6: Ejecución 3: Máximo 3 inserciones de jugadores por equipo, y 2 equipos por sede.

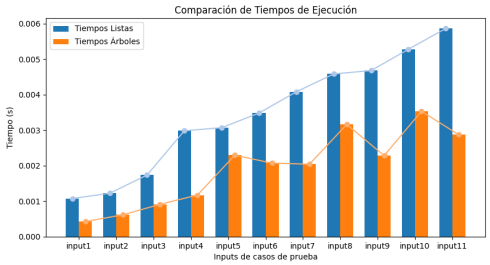


Figure 7: Ejecución 4: Máximo 3 inserciones de jugadores por equipo, y 2 equipos por sede.

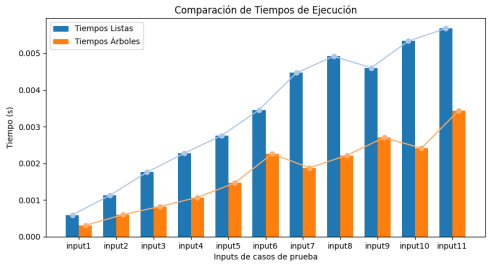


Figure 8: Ejecución 5: Máximo 3 inserciones de jugadores por equipo, y 2 equipos por sede.

Dichos resultados, evidencian el comportamiento esperado: sin embargo notamos algo interesante si modificamos la constante que indica el valor de jugadores máximo (además liberando el valor de inserción (de jugadores/equipos) de 3 a $\lfloor n/3 \rfloor$, en el caso de los equipos, y de 2 a $\lfloor n/6 \rfloor$ en el caso de las sedes), encontramos que ambas soluciones se ejecutan en tiempo mucho más cercano, como se puede ver a continuación:

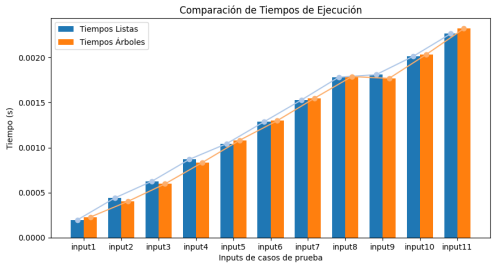


Figure 9: Ejecución 1: Máximo 100=n/3 inserciones de jugadores por equipo, y n/6 equipos por sede.

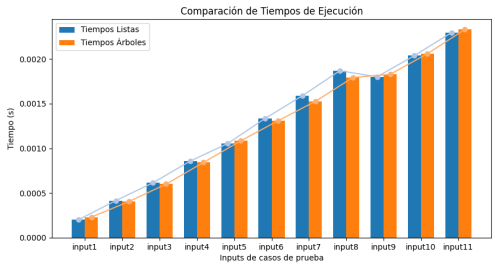


Figure 10: Ejecución 2: Máximo 100=n/3 inserciones de jugadores por equipo, y n/6 equipos por sede.

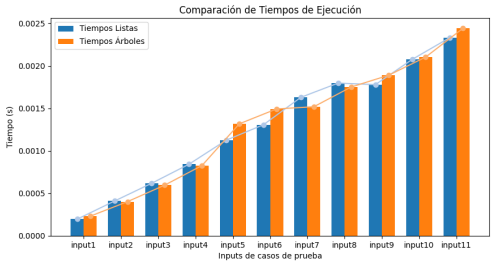


Figure 11: Ejecución 3: Máximo 100=n/3 inserciones de jugadores por equipo, y n/6 equipos por sede.

Por lo tanto, se evidencia que los algoritmos basados en arboles rojinegro son mas eficientes que los basados en listas sin comparaciones, en la mayoría de los casos. Casos donde se puedan presentar escenarios razonables en el sentido del problema que se desea modelar, se resalta la diferencia de la complejidad $O(n)$ y $O(\log n)$.

5.1 Resultados Prácticos

- **Counting Sort**: En la práctica, el algoritmo de Counting Sort mantuvo un comportamiento lineal en todas las instancias en que fue probado. Fue inferior a la solución basada en árboles rojinegros en la mayoría de los casos, pero alcanzando a éste en el último escenario: cuando la cantidad de jugadores por equipo tuviera la posibilidad de ser a lo más $\lfloor n/3 \rfloor$ y equipos por sede fueran $\lfloor n/6 \rfloor$.
- **Bucket Sort**: Este algoritmo presentó un rendimiento muy eficiente cuando los datos se distribuían de manera uniforme entre los buckets. Dicho comportamiento se adicionó al encontrado por Counting Sort, y entre ellos se mantuvo la linealidad dentro de un margen esperado.
- **Árboles rojinegros**: Logró evidenciarse que la implementación basada en árboles rojinegros superó considerablemente a los algoritmos anteriores, en la mayoría de los casos. Dicho comportamiento se mantuvo con la misma complejidad ya presentadas.

5.2 Comparación con la Teoría

- **Counting Sort**: La teoría predice un rendimiento lineal O(n + k), lo cual se cumplió en la práctica para rangos de valores limitados. No obstante, se observó un incremento en el tiempo de ejecución cuando k aumentaba significativamente, lo cual es consistente con las expectativas teóricas.
- **Bucket Sort**: Según la teoría, Bucket Sort debería funcionar en tiempo O(n) en el mejor de los casos. En la práctica, este comportamiento se observó claramente con una distribución uniforme de los datos. La

desviación de este rendimiento ideal en distribuciones no uniformes también fue predecible y consistente con la teoría.

- **Árboles rojinegros**: demostraron un comportamiento consistente con su complejidad computacional teórica, manteniéndose dentro del límite de O(log(n)) sin importar el tamaño de los datos de entrada.

Nota: Cabe resaltarse que el rendimiento fue estudiado en ambas soluciones para las operaciones de inserción y de ordenamiento, las cuales se estiman de mayor costo computacional; con el fin de comparar ambos algoritmos de manera general.

6 Conclusión del Proyecto

- La elección de algoritmos de ordenamiento y estructuras de datos para la resolución de problemas no es trivial. Es fundamental conocer la naturaleza y las necesidades que presenta la problemática con el objetivo de seleccionar y emplear las herramientas que permitan obtener una solución más eficaz.
- Aunque al resolver un problema se obtengan los mismos resultados sin importar el algoritmo de ordenamiento o la estructura de datos utilizada, estas soluciones pueden variar significativamente en cuanto a consumo de recursos y tiempo de ejecución.
- Resolver un problema no implica únicamente obtener los resultados deseados, sino también realizarlo mediante un proceso que considere las prioridades esenciales, como el tiempo, los recursos y el costo económico.
- La forma en que se implementaron las soluciones, manteniendo modularidad, conjunto al uso del paradigma orientado a objetos, ayudó a encontrar flexibilidad al momento de hacer pruebas y hacer cambios que lograsen integrar un código legible y extensible.
- Los algoritmos Counting Sort y Bucket Sort, trabajando en conjunto se mantuvieron acordes a la complejidad de tiempo lineal, en todos los casos, evidenciando su robustez.
- Al aumentar el número de jugadores que podían ser insertados a los equipos, se notó un impacto en el tiempo de ejecución de varios órdenes de magnitud en ambas soluciones, lo cual fue esperado, ya que el tratar más jugadores aumenta el costo operacional de las distintas ejecuciones de ordenamiento.