Universidad EAFIT

Estructura de datos y algoritmos I

Documentación Parcial 2 – Momento 2

Ismael García Ceballos

Carlos Alberto Álvarez Henao

Septiembre 21 de 2025

Punto 1: multiplicación de matrices:

Enfoque utilizado:

Para resolver este punto se implementaron tres versiones del algoritmo de multiplicación de matrices cuadradas:

- **matmul_base (i-j-k):** versión clásica de tres bucles anidados, fácil de entender, pero poco eficiente en matrices grandes.
- **matmul_ikj (i-k-j):** versión que mejora la localidad de memoria al recorrer primero filas de A y luego columnas de B.
- **matmul_blocked (tiling, BS=64):** versión optimizada que divide las matrices en bloques de tamaño fijo para aprovechar mejor la memoria caché y reducir accesos lentos a memoria.

Inicialmente ya se tenía implementada la generación de matrices aleatorias y la versión base del algoritmo (i-j-k). Posteriormente, con ayuda de **ChatGPT (OpenAI, 2025)** se añadieron:

- La variante **tiling**.
- La función genérica **medir** usando la librería <chrono> para calcular tiempos de ejecución.
- La recomendación de compilar con el comando:

g++ -O3 -march=native -funroll-loops main.cpp -o matmul

Esta forma de compilación permitió una reducción muy significativa en los tiempos de ejecución, ya que activa optimizaciones avanzadas:

- -O3: optimización agresiva del código.
- -march=native: usa instrucciones específicas de la CPU.
- -funroll-loops: desenrolla bucles, reduciendo overhead en operaciones repetitivas.

Referencias: https://chatgpt.com/

Dificultades encontradas:

- En el enunciado original se solicitaban funciones adicionales relacionadas con:

- **Validación de resultados:** comparar la matriz obtenida con una referencia para asegurar exactitud numérica.

- **Cálculo del speedup automáticamente en el código:** mostrar cuánto más rápida es cada variante respecto a la base.
- **Uso de valores promedio o mediana en las mediciones:** para reducir variaciones en los tiempos.

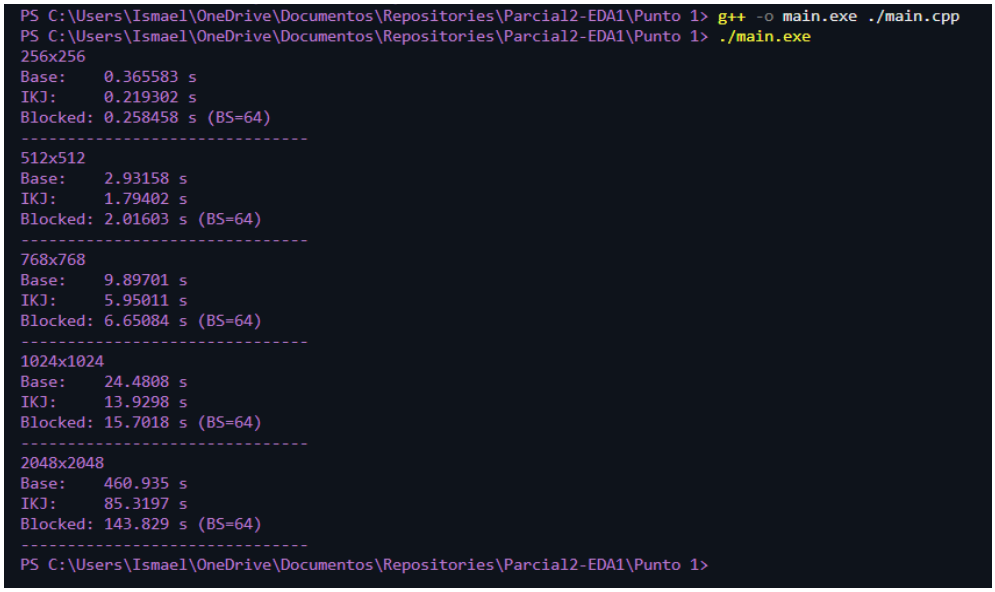
Estas funciones no se implementaron porque inicialmente no sabía cómo hacerlo. Al igual que con el tiling, pedí ayuda a ChatGPT (https://chatgpt.com/), pero en este caso no logré comprender completamente las explicaciones sobre validación y estadísticas de tiempos. Por esta razón, opté por no incluirlas y centrarme únicamente en las tres implementaciones principales (base, ikj y bloqueada), más la medición básica de tiempo.

- La versión base (i-j-k) resultaba demasiado lenta en tamaños grandes como 1024×1024 o 2048×2048 (ejecuciones de varios minutos).
- Se necesitó comprender cómo el orden de los bucles afecta el acceso a la memoria y la eficiencia del programa.
- La implementación del tiling requirió especial cuidado al manejar los límites de los bloques con min(...).

Estrategias de pruebas:

Se ejecutaron las tres versiones para tamaños de matrices crecientes:

- 256×256
- 512×512
- 768×768
- 1024×1024
- 2048×2048



En cada caso se midió el tiempo de ejecución en segundos. Se compararon las tres variantes para analizar la mejora en rendimiento.

Resultados obtenidos:

- Con la compilación básica (g++ main.cpp -o main.exe) los tiempos fueron muy altos:

  Ejemplo: para **2048×2048**, la versión base tomó más de 460 segundos.

- Con la compilación optimizada (g++ -O3 -march=native -funroll-loops), los tiempos se redujeron drásticamente:

  Para **2048×2048**, la versión base bajó a ~**16 s**, y la bloqueada a ~**1.2 s**.

- En todos los tamaños de prueba se verificó una clara mejora:

  IKJ fue más rápido que Base.

  Blocked (BS=64) fue el más eficiente en la mayoría de casos.

En conclusión, se logró mostrar cómo el orden de bucles y el uso de bloques optimizan la multiplicación de matrices, y cómo las opciones de compilación influyen directamente en el rendimiento.

Punto 2: Binary Search Tree (BST):

Enfoque Utilizado:

Para resolver este ejercicio se implementó una clase BST en C++ con las operaciones fundamentales de un árbol binario de búsqueda.

- **Estructura Node**: contiene un entero key y dos punteros (left y right) para los hijos izquierdo y derecho.
- **insert(key)**: inserta un nuevo valor respetando el invariante del BST (valores menores a la izquierda, mayores a la derecha). Para los duplicados se definió insertarlos siempre en el subárbol derecho.
- **find(key, camino)**: busca una clave en el árbol, devolviendo si existe y el recorrido desde la raíz hasta el nodo. El recorrido se almacena en un vector<int> para mostrar el camino.
- **Recorridos clásicos**:
    - *Inorder*: muestra los valores en orden creciente.
    - *Preorder*: muestra primero la raíz, luego subárbol izquierdo y derecho.
    - *Postorder*: muestra primero los hijos y al final la raíz.
- **altura()**: devuelve la altura del árbol medida en aristas.
- **gradoMax()**: calcula el grado máximo de los nodos del árbol (puede ser 0, 1 o 2).
- La salida de todos los recorridos y caminos utiliza el formato dato --> ... --> NULL para mayor claridad.

Dificultades encontradas:

- La primera dificultad fue definir una política para duplicados, ya que por defecto un BST no los contempla. Se optó por insertarlos siempre en el subárbol derecho.
- Definición de la altura del árbol: inicialmente se implementó contando nodos, pero luego se corrigió para contar aristas, de modo que una hoja tiene altura = 0 y un árbol vacío altura = -1.
- Se ajustó el formato de impresión para que los números aparecieran separados con --> y terminaran en NULL.

Estrategias de prueba:

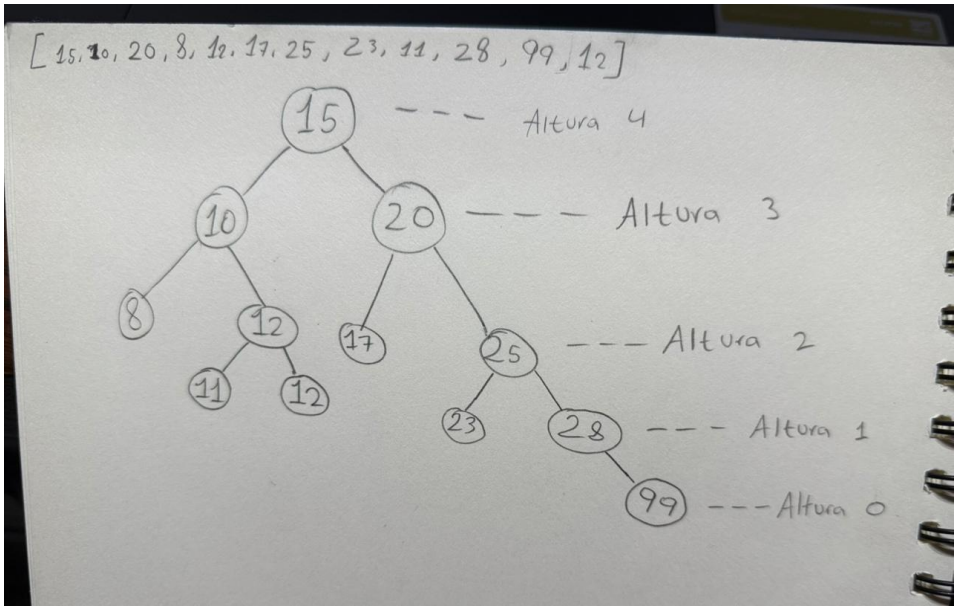
Se diseñaron diferentes casos de prueba, cambiando ligeramente la función main del código, para validar el correcto funcionamiento del BST:

1. **Caso base**: insertar la lista [15, 10, 20, 8, 12, 17, 25] (propuesta en el documento) y mostrar los recorridos, altura y grado máximo.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> g++ -o main.exe ./main.cpp
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> ./main.exe
Recorrido Inorder: 8 --> 10 --> 12 --> 15 --> 17 --> 20 --> 25 --> NULL
Recorrido Preorder: 15 --> 10 --> 8 --> 12 --> 20 --> 17 --> 25 --> NULL
Recorrido Postorder: 8 --> 12 --> 10 --> 17 --> 25 --> 20 --> 15 --> NULL
Altura: 2
Grado maximo: 2
Buscando 12: Encontrado. Camino: 15 --> 10 --> 12 --> NULL
Buscando 30: No encontrado. Camino recorrido: 15 --> 20 --> 25 --> NULL
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2>
```

2. **Caso con claves adicionales**: insertar [23, 11, 28, 99, 12] para verificar la inserción de nuevos valores y duplicados.



3. **Búsquedas exitosas**: por ejemplo, buscar 12 y 99, mostrando el camino desde la raíz.
4. **Búsquedas fallidas**: por ejemplo, buscar 30, mostrando el camino recorrido hasta detectar la ausencia.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> g++ -o main.exe ./main.cpp
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> ./main.exe
Recorrido Inorder: 8 --> 10 --> 11 --> 12 --> 12 --> 15 --> 17 --> 20 --> 23 --> 25 --> 28 --> 99 --> NULL
Recorrido Preorder: 15 --> 10 --> 8 --> 12 --> 11 --> 12 --> 20 --> 17 --> 25 --> 23 --> 28 --> 99 --> NULL
Recorrido Postorder: 8 --> 11 --> 12 --> 12 --> 10 --> 17 --> 23 --> 99 --> 28 --> 25 --> 20 --> 15 --> NULL
Altura: 4
Grado maximo: 2
Buscando 12: Encontrado. Camino: 15 --> 10 --> 12 --> NULL
Buscando 30: No encontrado. Camino recorrido: 15 --> 20 --> 25 --> 28 --> 99 --> NULL
Buscando 99: Encontrado. Camino: 15 --> 20 --> 25 --> 28 --> 99 --> NULL
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2>
```

5. **Caso degenerado**: insertar una lista ordenada (ejemplo [1,2,3,4,5,6,7]), lo que convierte el árbol en una “lista encadenada”, con altura máxima.

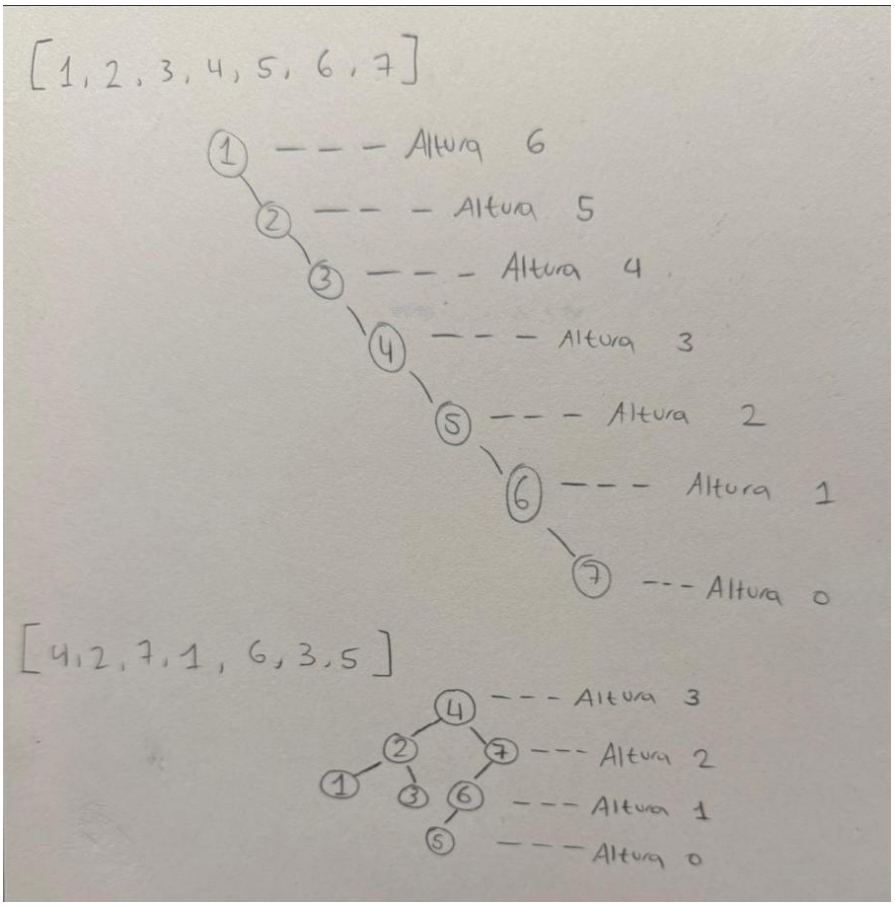
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> g++ -o main.exe ./main.cpp
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> ./main.exe
Recorrido Inorder: 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> NULL
Recorrido Preorder: 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> NULL
Recorrido Postorder: 6 --> 5 --> 4 --> 3 --> 2 --> 1 --> NULL
Altura: 5
Grado maximo: 1
Buscando 4: Encontrado. Camino: 1 --> 2 --> 3 --> 4 --> NULL
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2>
```

6. **Caso aleatorio**: insertar valores en orden aleatorio para comparar la altura con el caso degenerado.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> g++ -o main.exe ./main.cpp
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> ./main.exe
Recorrido Inorder: 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> NULL
Recorrido Preorder: 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> NULL
Recorrido Postorder: 7 --> 6 --> 5 --> 4 --> 3 --> 2 --> 1 --> NULL
Altura: 6
Grado maximo: 1
Buscando 4: Encontrado. Camino: 1 --> 2 --> 3 --> 4 --> NULL
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> g++ -o main.exe ./main.cpp
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2> ./main.exe
Recorrido Inorder: 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> NULL
Recorrido Preorder: 4 --> 2 --> 1 --> 3 --> 7 --> 6 --> 5 --> NULL
Recorrido Postorder: 1 --> 3 --> 2 --> 5 --> 6 --> 7 --> 4 --> NULL
Altura: 3
Grado maximo: 2
Buscando 4: Encontrado. Camino: 4 --> NULL
PS C:\Users\Ismael\OneDrive\Documentos\Repositories\Parcial2-EDA1\Punto 2>
```



Resultados obtenidos:

- El árbol construido con [15, 10, 20, 8, 12, 17, 25, 23, 11, 28, 99, 12] mostró recorridos correctos, validando que el inorden devuelve siempre los valores en orden creciente, incluyendo el duplicado 12.
- La función find imprimió correctamente los caminos raíz→clave, diferenciando entre claves encontradas y no encontradas.
- La altura se calculó correctamente en términos de aristas: el camino más largo (15 → 20 → 25 → 28 → 99) dio altura 4.
- El grado máximo de los nodos fue 2, lo cual corresponde a un árbol binario.
- En el caso degenerado con inserción ordenada, la altura fue máxima (n-1), mientras que con inserción aleatoria la altura resultó significativamente menor, mostrando la diferencia en eficiencia.
- Los recorridos y búsquedas se imprimieron en el formato dato --> ... --> NULL, cumpliendo con el estándar de legibilidad propuesto.