



## EC-200 Data Structures

---

### Project Report for “File Storage System”

**SUBMITTED TO:**

Lecturer Anum Abdul Salam

**SUBMITTED BY:**

**Student Name:** Tazeen ur Rehman

**Reg #** 466342

**Student Name:** Rida Zahra

**Reg #** 465791

**Student Name:** Mawa Chaudhary

**Reg #** 474121

DE-45

**Submission Date:** 11-05-202

## Contents

Introduction :.....	3
<b>Virtual Harddisk:</b> .....	3
Problem Statement: .....	4
Objectives: .....	4
System Design: .....	5
Class Diagram .....	5
Implementation Details: .....	6
<b>Programming language Used:</b> .....	6
<b>Calculations and Distribution of the file “File_System” :</b> .....	6
<b>File Directory List(1MB):</b> .....	6
<b>AddressMinHeap(1MB):</b> .....	6
<b>File Content(8MB):</b> .....	7
<b>Application of Data Structure principles :</b> .....	7
<b>Key classes and methods :</b> .....	8
<b>Choice of Data Structures :</b> .....	12
Key features: .....	16
Results and Outputs: .....	22
Challenged and Solutions: .....	24
Conclusion: .....	24
External Tools:.....	25

## **Introduction :**

### **Virtual Harddisk:**

This project presents the development of a Virtual Hard Disk (VHD) file storage system—a simulated storage environment that mimics the functionality of a physical hard drive. It provides a practical and risk-free platform for understanding how modern file systems store, manage, and access data. The aim of the project is to replicate core operations of real hard disks using a custom-built file system on top of a virtual storage environment.

A Virtual Hard Disk is essentially a large file stored on a physical device that emulates an actual hard drive. It allows data to be stored and managed independently, enabling a system to organize files as if they were on a separate storage device. In this project, a 10MB file named "File\_system" acts as the virtual hard disk, serving as a sandbox for implementing and testing file system functionality without affecting the host operating system.

The virtual hard disk file storage system is significant because it provides insight into how data is handled at the operating system level. File systems are critical to computer systems—they define how files are named, stored, retrieved, and protected. Without them, data management would be chaotic, unreliable, and inefficient. By simulating a file system within a virtual hard disk, this project highlights how storage devices maintain order, integrity, and accessibility of data.

Built on top of the VHD, the implemented File System in this project includes the following key features:

- **Data Organization:** Files are arranged in a hierarchical structure of directories and subdirectories.
- **Metadata Management:** Each file retains essential attributes such as name, size, storage location, and access permissions.
- **Storage Allocation:** The system efficiently tracks used and free space, allowing for optimal allocation and deallocation.
- **File Access:** Users can create, read, write, modify, and delete files as needed.

To further enhance usability and mimic real system behavior, the project supports several advanced operations:

- Adding new files to the virtual disk
- Deleting existing files
- Viewing files and directory structure
- Copying files from the virtual disk to the Windows file system
- Copying files from Windows into the virtual disk
- Modifying file contents
- Defragmenting the virtual disk to optimize space and performance

Overall, this project serves as a practical tool to understand how file systems operate at a low level. It applies fundamental data structures and file handling techniques to simulate the core behavior of modern storage systems, making it an effective learning model for computer science and software engineering students.

## **Problem Statement:**

Efficient file management is a fundamental requirement in any operating system. Real-world file systems must not only store and retrieve files but also handle metadata, manage storage addresses, and maintain performance as the volume of data grows. However, experimenting directly on physical storage devices poses risks such as data corruption or system failure. This limits students and developers from gaining hands-on experience with the underlying mechanisms of file systems.

This project addresses the problem by simulating a simplified Virtual Hard Disk (VHD) and File System that replicate the basic functionalities of real-world systems—such as adding, deleting, modifying, copying, and defragmenting files—within a 10MB virtual environment. A key challenge solved by this system is efficient file management with proper address tracking and defragmentation, which ensures optimal space usage and performance.

An important aspect of the project is the choice of appropriate data structures for storing and managing file metadata (e.g., file names, sizes, block addresses, and permissions). Selecting efficient structures (such as hash tables, arrays, or linked lists) directly impacts the time complexity of operations like searching, updating, and allocating storage blocks. Faster access to metadata leads to smoother file system performance, especially as the number of files increases. By modeling how a file system stores metadata, tracks free and occupied space, and optimizes performance through defragmentation, this project demonstrates the critical role of algorithmic efficiency and data structure design in system-level software. It provides learners with a deep understanding of the trade-offs between memory usage and speed, and how these trade-offs influence the design of real storage systems.

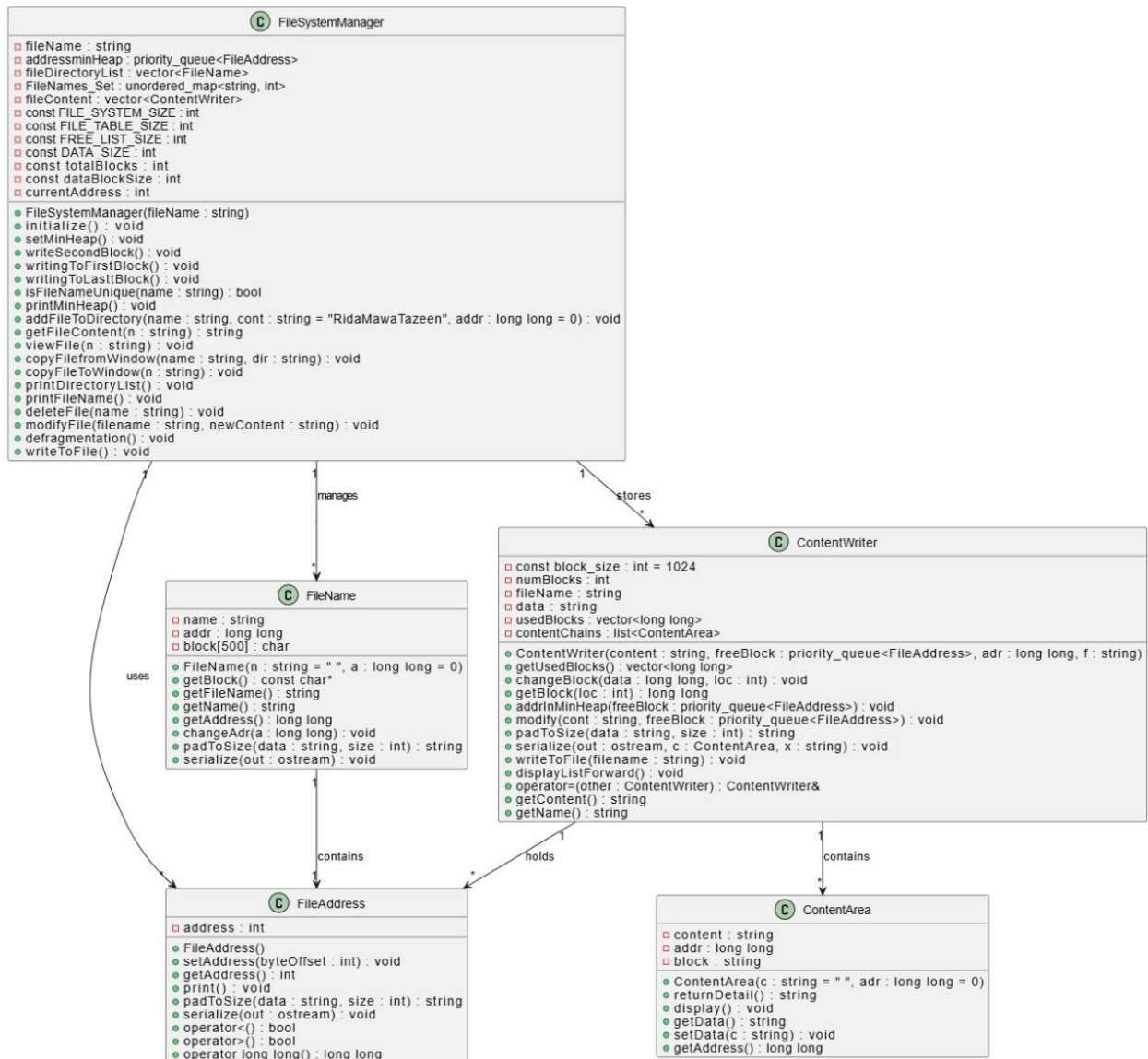
## **Objectives:**

- To develop a Virtual Hard Disk (VHD) that emulates the functionality of a physical hard drive for practical understanding of file system operations.
- To design a custom-built file system on top of the VHD for replicating the core operations of real storage systems such as file creation, modification, deletion, and defragmentation.
- To implement efficient metadata management by storing attributes like file name, size, storage location, and permissions for each file in the system.
- To detect and implement the most efficient data structure for each part of the 10MB virtual file to optimize metadata management, including choosing between minheaps, arrays, or linked lists for various tasks based on time complexity considerations.
- To minimize time complexity for common file system operations (e.g., searching, updating, and allocating storage blocks) by applying the most suitable data structures that enhance performance as the volume of files grows.
- To incorporate storage allocation strategies that track free and used space, ensuring optimal allocation and deallocation of storage resources.

- To support advanced file operations such as copying files between the virtual disk and the host file system, modifying file contents, and viewing the directory structure.
- To demonstrate the process of defragmentation to optimize storage space and improve system performance.

## System Design:

### Class Diagram



## Implementation Details:

### **Programming language Used:**

- C++

## Calculations and Distribution of the file “File System” :

### File Directory List(1MB):

The first 1 MB of the File\_System file is reserved for metadata, including file names and their addresses. This area spans:

- Start Address: 0 Bytes
- End Address: 1,048,575 Bytes
- End Address: 1,048,575 Bytes
- Total Size: 1 MB=1024×1024=1,048,576 Bytes
- Each entry (or **block**) in the directory is **500 Bytes wide**, so the **total number of directory blocks** is:

$$\frac{1,048,576 \text{ Bytes}}{500 \text{ Bytes/Block}} = 2097.152 \approx 2097 \text{ blocks}$$

- This implies that a **maximum of 2,097 files** can be stored in the file system, since **each file must have a unique name entry** in the directory.

### AddressMinHeap(1MB):

- The second 1 MB of the File\_System file is dedicated to storing free block addresses in a min-heap format, allowing efficient allocation of the smallest free block. This area spans:
- Start Address: 1,048,576 Bytes
- End Address: 2,097,151 Bytes
- Total Size:
- 1 MB=1,048,576 Bytes
- This section corresponds to managing the **8 MB content area** (detailed below), which is divided into **8192 blocks**. Therefore, the min-heap must be able to hold up to **8192 addresses**. The **size of each address entry** in this heap is:

$$\frac{1,048,576 \text{ Bytes}}{8192 \text{ Blocks}} = 128 \text{ Bytes per address}$$

**File Content(8MB):**

The **third section** of the file is the **File Content Area**, where the actual file data is stored. This section spans:

- **Start Address:** 2,097,152 Bytes
- **End Address:** 10,485,760 Bytes
- **Total Size:** 8 MB=8×1024×1024=8,388,608 Bytes
- Each block in this area is fixed at **1,024 Bytes** (1 KB), allowing:

$$\frac{8,388,608 \text{ Bytes}}{1024 \text{ Bytes/Block}} = 8192 \text{ blocks}$$

- Thus, the system can store file content in **up to 8192 blocks**, each independently addressable and managed via the min-heap.

**Application of Data Structure principles :****1. Purposeful Selection of Data Structures**

In this implementation, a variety of data structures were carefully selected to fulfill specific roles based on their strengths. A `vector<FileName>` is used to store file metadata—such as names and addresses—in a sequential manner, making it efficient for appending new entries with  $O(1)$  time complexity. Although searching within this vector is  $O(n)$ , the impact is mitigated by the use of an `unordered_map<string, int>` called `FileNames_Set`, which offers constant time ( $O(1)$ ) average-case lookups to verify the existence of a file. Additionally, a `priority_queue<FileAddress, vector<FileAddress>, greater<FileAddress>>` functions as a min-heap, allowing the system to efficiently manage and retrieve the smallest available free address block with  $O(\log n)$  time for insertion and deletion.

**2. Time Complexity Efficiency**

The use of an `unordered_map` drastically reduces the overhead of checking file existence by providing  $O(1)$  access time. Similarly, the `priority_queue` ensures optimal memory block allocation using its log-scale performance for insertions and deletions. To handle file contents logically, a `vector<ContentWriter>` is employed, benefiting from constant time appends and direct index-based access. While writing file data inherently requires  $O(n)$  time—where  $n$  is the size of the content—this is considered acceptable due to the nature of data input/output operations. Overall, the selection and combination of these data structures reflect a thoughtful emphasis on time complexity and runtime efficiency.

**3. Design Considerations and Future Extensibility**

Beyond raw performance, the implementation also shows attention to design principles such as memory efficiency and modularity. Memory is allocated and padded only when necessary, reducing overhead. Functions like `seekg`, `seekp`, `read`, and `write` are utilized to maintain fast and direct binary disk I/O operations.

Moreover, the code demonstrates object-oriented encapsulation by logically dividing responsibilities into classes like `FileName`, `FileAddress`, and `ContentWriter`, making the system more maintainable and extensible. Supporting structures like stack and list are also present—though not yet used—indicating potential for future features like undo functionality or directory hierarchy management.

## Key classes and methods :

### 1. FileSystemManager:

This is the main class that manages the file system, including file storage, file directory, and content management.

#### Methods:

- **initialize():**
  - This method checks if the file exists and is empty. If it is empty, it sets up the file system with a predefined file size and prepares a min-heap for available blocks. If the file exists and contains data, it loads the file directory and content from the file system.
  - **Key Concepts:** File padding, reading binary files, managing free space using min-heap.
- **setMinHeap():**
  - Populates the priority queue with available blocks for file storage, starting from a specific address and adding blocks sequentially.
  - **Time Complexity:**  $O(n)$ , where  $n$  is the total number of blocks (8192 in this case).
- **writeSecondBlock() and writingToFirstBlock():**
  - These methods write the file system's structure (file names and addresses) to the first and second blocks of the file. The content is serialized and written using the `serialize()` method from the `FileAddress` and `ContentWriter` classes.
  - **Time Complexity:**  $O(n)$ , where  $n$  is the number of files and content entries.
- **addFileToDirectory():**
  - Adds a new file to the directory if the name is unique. The method pushes the file's address and content to the respective structures (min-heap, file directory list, and content list).
  - **Time Complexity:**  $O(1)$  for checking the name uniqueness using the unordered map and  $O(1)$  for insertion in the vector.
- **getFileContent():**
  - Retrieves the content of a file by its name. The file's content is fetched from the `fileContent` vector based on the file name's index in the unordered map.
  - **Time Complexity:**  $O(1)$  for accessing the content of the file.
- **viewFile():**
  - Displays the content of a file if it exists.
  - **Time Complexity:**  $O(1)$  for checking existence and retrieving content.



- **copyFilefromWindow():**
  - Copies a file from the local system into the file manager. It reads the file content and adds it to the file directory.
  - **Time Complexity:**  $O(n)$ , where  $n$  is the size of the file being copied.
- **copyFileToWindow():**
  - Copies the file content from the file system to the local disk. It retrieves the content from the file system and writes it to a local file.
  - **Time Complexity:**  $O(n)$ , where  $n$  is the size of the file being copied.

## 2. FileName

- **const char getBlock() const\***
  - Returns a pointer to the block array. This can be used for direct binary storage or memory manipulation.
- **string getFileName() const**
  - Combines the name and addr into a single string with a space between them. Useful for display or serialization purposes.
- **string getName() const**
  - Returns the file name stored in the name variable.
- **long long getAddress() const**
  - Returns the stored file address (addr).
- **void changeAdr(long long a)**
  - Updates the addr value to the provided value a.
- **string padToSize(const string& data, int size) const**
  - If data is shorter than size, it pads it with spaces. If it's longer, it trims it. This ensures uniform length of serialized output.
- **void serialize(ostream& out) const**
  - Converts the file name and address into a single string padded to 500 characters using padToSize(), and writes it as a binary stream. This ensures consistent storage format across records.

## 3. FileAddress

### Methods:

- **FileAddress()**
  - This is the default constructor that initializes the address to -1, indicating an invalid or uninitialized state.

- **setAddress(int byteOffset)**
  - o Sets the internal address to the provided byteOffset, allowing the object to represent a specific byte offset in a file.
- **getAddress() const**
  - o Returns the current value of the address. Being a const method ensures that it does not alter the object state.
- **print() const**
  - o Prints the value of address followed by a newline. This is mainly for debugging or displaying the internal address.
- **padToSize(const string& data, int size) const**
  - o Ensures a string is exactly size characters: it appends spaces if the string is shorter, or trims characters if it's longer. Used to maintain fixed-size formatting for file writing.
- **serialize(ostream& out) const**
  - o Converts the address to a string, pads it to 128 characters using padToSize(), and writes it to an output stream. This allows saving the object in a consistent, fixed-size format.
  - o Key Concepts: Object serialization, file I/O, fixed-size record format.
- **operator<(const FileAddress& f) const**
  - o Compares this object's address with another FileAddress object's address and returns true if it's smaller. Useful for sorting or inserting in ordered structures.
- **operator>(const FileAddress& f) const**
  - o Returns true if this object's address is greater than the other's. Complements the < operator for sorting and priority operations.
- **operator long long() const**
  - o Allows implicit conversion of a FileAddress object to a long long, returning the address as a large integer. Useful when interfacing with systems or functions expecting numeric types.

#### 4. ContentArea

##### Methods:

- **ContentArea(string c = " ", long long adr = 0)**
  - o Initializes the content and addr variables with the given values. Default values are provided to allow empty initialization.

- **string returnDetail() const**
  - Returns the content string followed by a space. This is useful for appending or preparing content for display or serialization.
- **void display() const**
  - Displays the content using cout, utilizing returnDetail() to format the string.
- **string getData() const**
  - Returns the current content stored in the content variable.
- **void setData(string c)**
  - Updates the content variable with the given string c.
- **long long getAddress()**
  - Returns the stored address addr.

## **5. ContentWriter**

ContentWriter manages content writing into fixed-size blocks by breaking input data into ContentArea chunks, assigning them memory addresses, and writing them to file streams.

### **Methods:**

- **getUsedBlocks()**
  - Returns the list of block addresses used for storing the content.
- **changeBlock(long long data, int loc)**
  - Changes the address at the specified location in the usedBlocks vector.
- **getBlock(int loc)**
  - Returns the block address at the given location.
- **modify(string cont, priority\_queue<FileAddress, vector<FileAddress>, greater<FileAddress>>& freeBlock)**
  - Appends new content to the existing data. If the last block has space, it appends there; otherwise, it creates new blocks for remaining data using free addresses.
- **padToSize(const string& data, int size)**
  - Pads or trims a string to fit exactly size characters.
- **serialize(ostream& out, const ContentArea& c, string x)**
  - Prepares and writes the serialized ContentArea data along with a link/address (x) to the output stream.

- **writeToFile(const string& filename)**
  - Opens a binary file, writes each ContentArea to its corresponding address, and connects blocks through x.
- **displayListForward() const**
  - Iterates through contentChains and displays each content block.
- **ContentWriter& operator=(const ContentWriter& other)**
  - Assignment operator to copy the file name, data, used blocks, and content chains from another ContentWriter object.
- **getContent() const**
  - Returns the complete content string.
- **string getName() const**
  - Returns the file name.

## Choice of Data Structures :

### 1. Data Structure for FileDirectorylist(First 1MB):

#### First Consideration:

Filenames were to be stored along with starting address of each file one after another .The first Data structure we chose was LinkedList for it.

#### Pros of LinkedList:

- Linked lists can grow or shrink in size at runtime without needing to define a maximum size in advance (no memory wastage or reallocation).
- Inserting or deleting nodes (especially in the middle or beginning) is faster in linked lists—no need to shift elements like in arrays.
- linked lists do not need a block of contiguous memory, making memory allocation easier in fragmented memory environments.

#### Cons of LinkedList:

- Cannot access a file by index (e.g., list[5]) in constant time; traversal is required from the head each time, making search operations inefficient.
- One file pointing to another (via next pointer) doesn't represent real-world relationships in a directory; files are not naturally sequentially related.

- Many operations (search, locate, update) become **linear-time ( $O(n)$ )**, which is inefficient for directories with large numbers of files.
- Linked lists do not benefit from CPU caching due to non-contiguous memory allocation, leading to **slower traversal speeds** compared to arrays.
- Debugging and maintaining pointer integrity is harder, especially in large, real-time file systems where consistency is crucial.

Considering the way the LinkedList increased time complexity for the methods of File Directory List, it didn't seem a suitable data structure.

### **Final Consideration:**

We decided to use the Data Structure “Vectors” to handle the directory list as they offered better useability for the functions we needed to perform.

Pros:

- Vectors allow direct access to any element using an index (i.e., `vec[i]`) in constant time  $O(1)$ , while linked lists require  $O(n)$  traversal.
- Vectors store elements contiguously in memory, improving CPU cache performance and making iteration much faster than a linked list.
- Vectors only store the data (and a small internal buffer), while linked lists require extra memory for each pointer (next) in every node.
- Vectors automatically handle memory allocation and deallocation behind the scenes. In linked lists, you must manually manage memory for each node.
- Vectors are compatible with most Standard Template Library (STL) algorithms like `sort()`, `binary_search()`, `find()`, etc., unlike linked lists.

### **2. Data Structure for address area(Second 1MB):**

To maintain a sorted structure of available addresses, it was essential that every time a file was added, it would be stored at the first available address—which should always be efficiently retrievable. A min-heap was ideal for this purpose, as it consistently provides the smallest (i.e., lowest) address at the top. Similarly, whenever a file was deleted, its freed address needed to be reinserted into the pool of available addresses in sorted order. This requirement also extended to defragmentation, where efficient reallocation and address management were critical. Considering all these needs—fast retrieval of the minimum address, efficient insertion, and automatic sorting—the min-heap emerged as the most suitable data structure, and was therefore chosen to manage the address space.

Pros:

- Always gives the minimum address in  $O(1)$  time, which is ideal when files must be stored at the lowest possible address.
- Maintains addresses in a sorted structure without requiring manual sorting after every insertion or deletion.
- Adding or removing addresses from the pool takes  $O(\log n)$  time, ensuring scalability for large systems.

- Freed addresses can be pushed back into the heap and naturally reused in sorted order, reducing fragmentation.
- Works well when the number of used and free blocks changes frequently, as it dynamically adjusts with each update.

### **3. Data Structure for a single file**

As directed, each file was to be stored in the content area using blocks of 1024 bytes. However, since a file could exceed 1024 bytes, multiple blocks were often required. In such cases, all blocks belonging to a single file needed to be logically linked. Initially, the file data could be stored in consecutive blocks, but with multiple insertions and file modifications over time, the newly added data might be saved in blocks that were no longer contiguous with the file's original blocks. This meant the file's data had to be stored in a non-contiguous manner.

When such a file needed to be displayed, it had to start from the beginning of the data and follow through all the blocks in the correct order—similar to how a linked list is traversed starting from the head node. Considering these requirements, a linked list proved to be the most efficient data structure for storing a file's content, as it naturally supports non-contiguous memory and maintains logical order through links. Therefore, we finalized Linked List as the data structure to represent file content.

Pros:

- Linked lists allow storing file data across non-contiguous memory blocks, which is ideal when blocks get fragmented over time.
- Files can grow or shrink without needing a predefined size or large continuous memory space, reducing memory wastage.
- Adding or removing blocks (e.g., when a file is modified or deleted) is efficient as it only involves updating pointers—no shifting of data is needed.
- Linked lists inherently support traversal from the start to the end, making it easy to read the file in the correct order, regardless of physical block locations.
- Unlike arrays or contiguous block allocation, linked lists prevent issues caused by external fragmentation in memory.

### **4. Data structure to handle Linked Lists of files:**

#### **First Consideration:**

Every time a file needed to be modified or deleted, it first had to be located efficiently, ideally with minimal time complexity. Initially, we considered using hash tables with a simple hash function that would consistently return the same unique index for a given key. While this approach was conceptually straightforward, implementing and maintaining an additional key attribute for each file—especially when files were being added, deleted, or modified randomly—proved inefficient. Despite the hashing, certain operations still required traversal or additional handling, which diminished its practical benefit in our case.

Cons:

- Hash tables do not preserve any order of entries. If files need to be listed in sorted or insertion order (as is often required in directories), additional effort is needed.
- Despite a good hash function, collisions are inevitable. Resolving them (via chaining or open addressing) adds complexity and can degrade performance.
- If the table is sparsely populated (due to poor distribution or a large capacity set for scalability), it can waste memory.
- Maintaining unique keys for files (especially if they change or need to be generated for each file dynamically) complicates file management.
- The efficiency of the hash table heavily depends on a well-designed hash function, which can be tricky to create and maintain.

**Final Consideration:**

Since an efficient and dynamic direct-addressing structure was needed, we opted for a hybrid approach. We used a Vector, where each index pointed to a linked list representing a file's data. Vectors enabled direct access, significantly reducing the time complexity for operations like addition and deletion.

To keep track of which index a file was stored at, we introduced an Unordered Map. This map used filenames as keys and returned the corresponding vector index, allowing us to retrieve any file in  $O(1)$  time. This combination of direct indexing and dynamic linkage made search operations highly efficient, and therefore, we finalized this hybrid structure for managing file contents.

Pros:

- Vectors allow constant-time random access, so once you know the index, reaching a file is extremely fast.
- The unordered map provides  $O(1)$  average time complexity for searching a file by name, solving the issue of linear traversal in linked lists.
- Vectors store elements in contiguous memory, improving cache performance for fast access to file references.
- This structure easily scales as the number of files increases, without a significant increase in lookup or access time.
- The **unordered map** allows quick  **$O(1)$  access** to the index of a file's linked list in the vector. This is extremely useful when identifying fragmented files and determining where their parts are stored. With this fast lookup, you can easily locate all blocks that belong to the same file, even if they are spread across non-contiguous memory locations.
- Since the **vector** provides a contiguous block of memory, you can rearrange fragmented blocks (i.e., blocks that are not placed consecutively) into consecutive slots within the vector. By leveraging the **unordered map** to track the file locations, you can ensure that all related blocks of a file are placed next to each other.

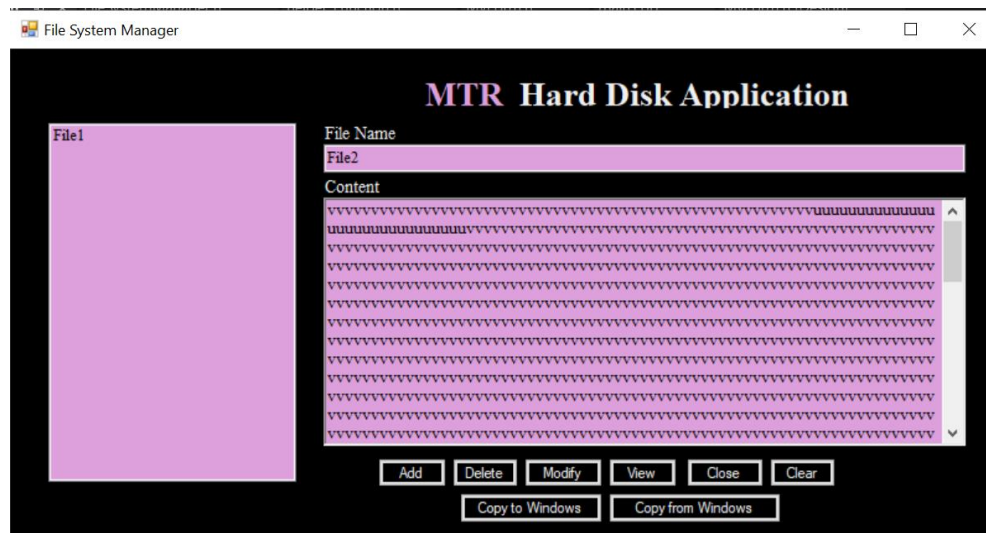
## Key features:

### 1. Initializing a file:

When executed, the application initializes by loading data from the 'FileSystem' file.

### 2. Adding a file:

When a user adds a file to the system, they enter the file name and its content, then press Enter. The file is then stored at the appropriate address within the system.

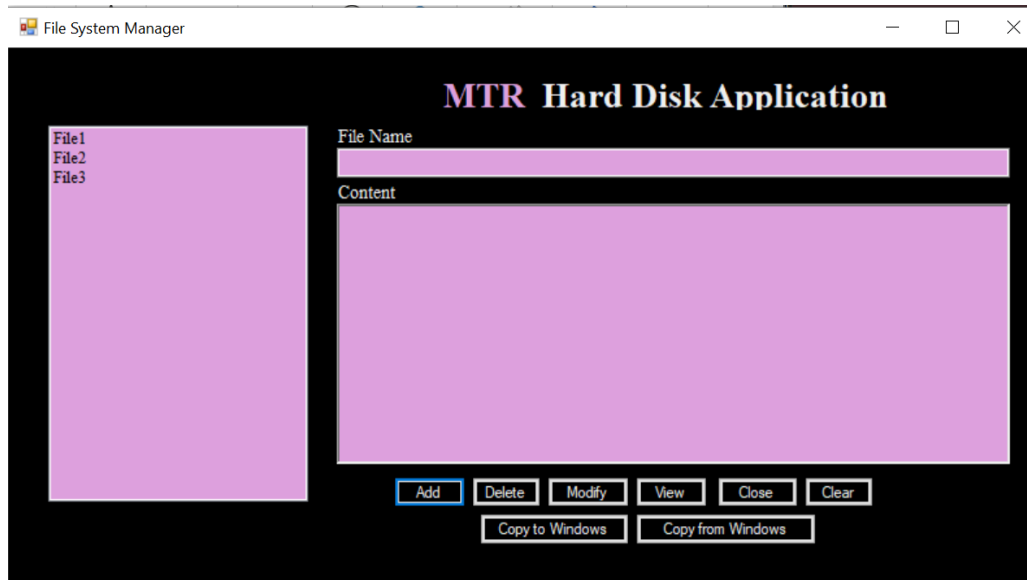


### 3. Deleting a File:

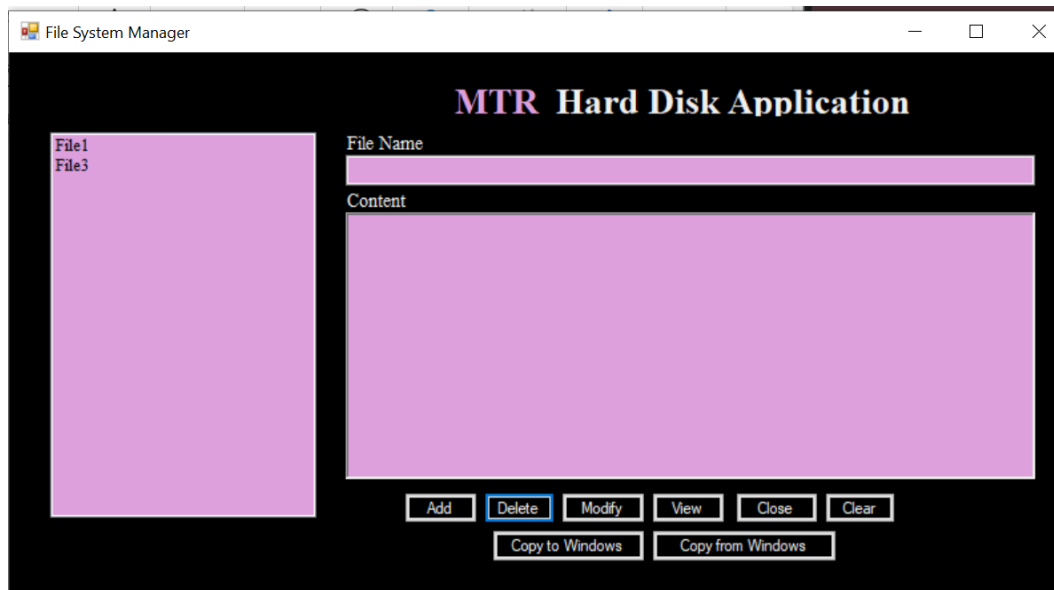
When a user chooses to delete a file, they simply select the file and press the delete button. The system then locates the file, releases all associated memory blocks, and updates the address management structure to reflect the freed space.



**Before Deletion:**

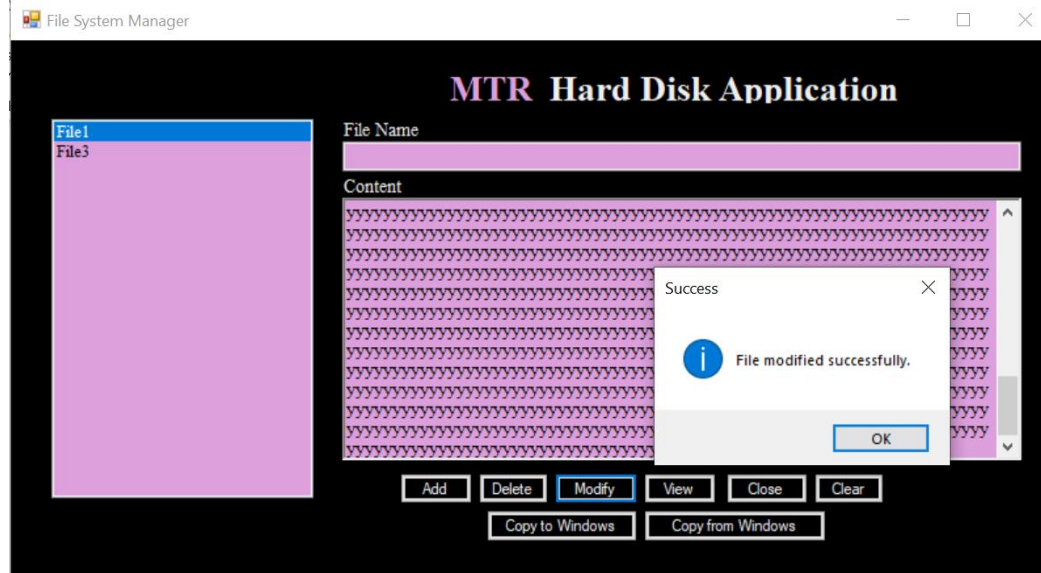


**After Deletion:**



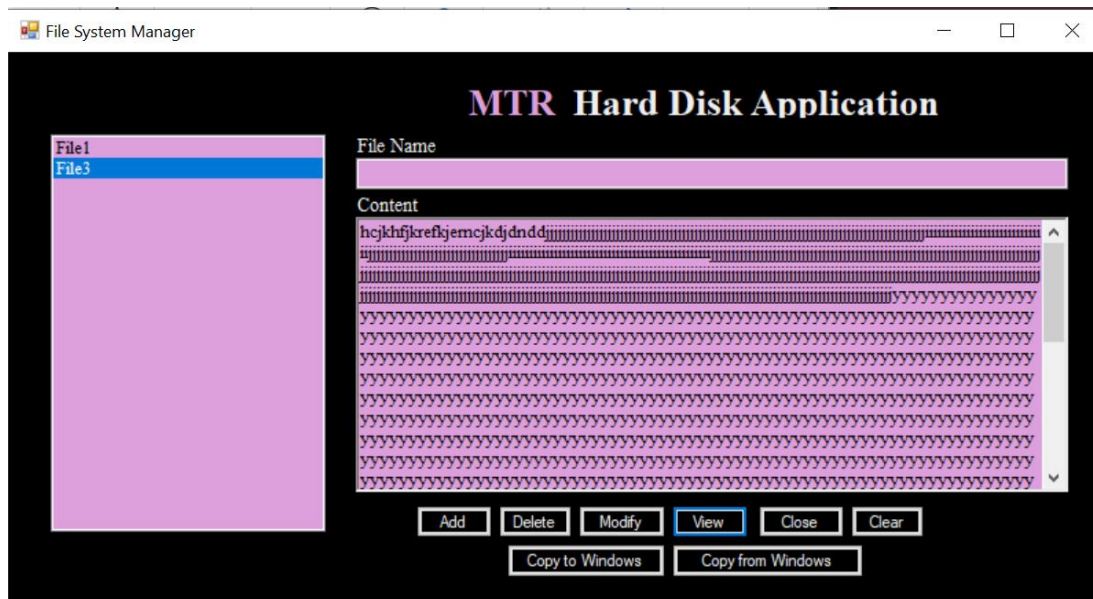
#### 4. Modifying a File:

To modify an existing file, the user selects the file and provides new or additional content. Upon pressing the *Modify* button, the system updates the file by appending or replacing the data as needed, while ensuring that block links and storage integrity are properly maintained.



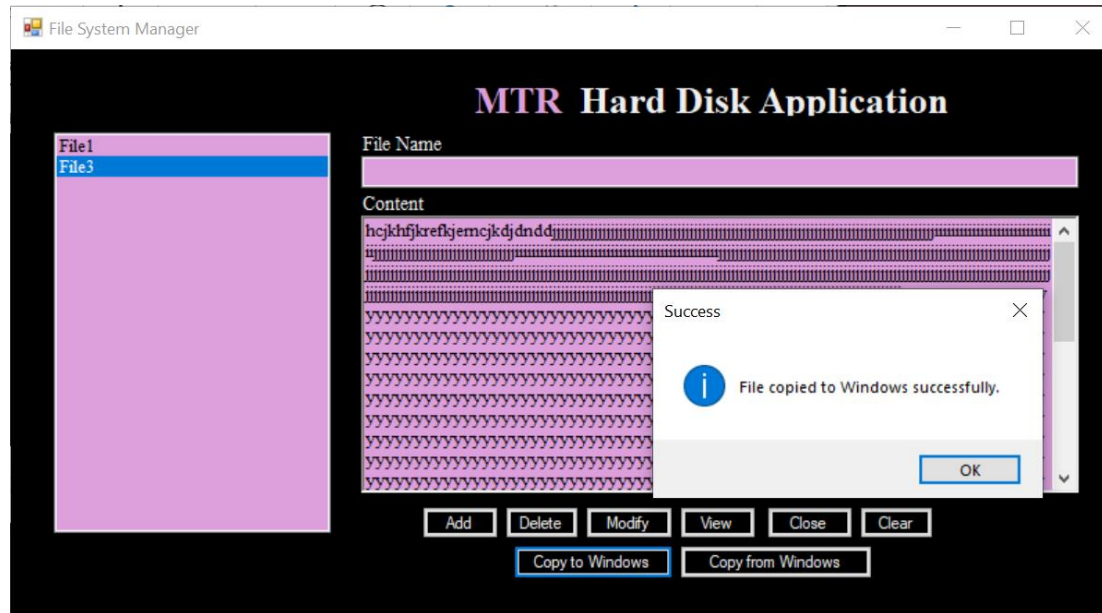
#### 5. Viewing a File:

When viewing a file, the user selects the file name. The system retrieves and displays the file content by traversing the linked blocks from start to end.



## 6. Copying a File to Windows:

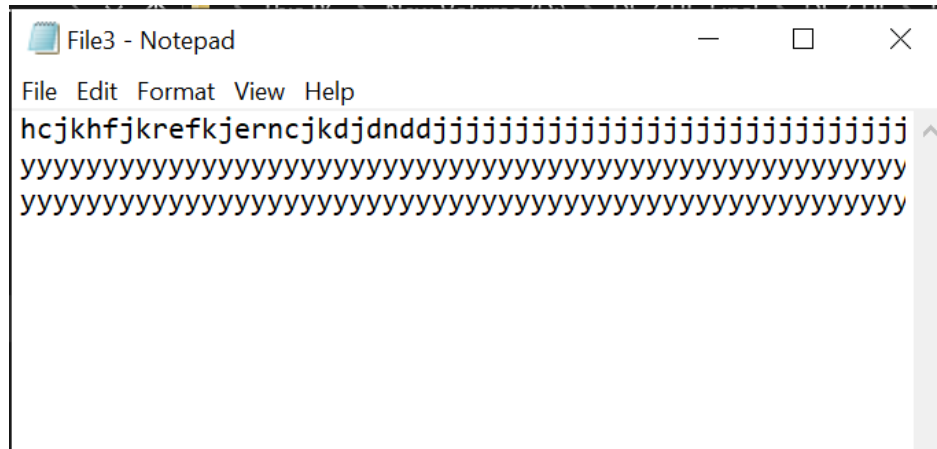
To copy a file from the application to the Windows file system, the user selects the file to export. The system reads its content and creates a corresponding file in the Windows directory.



File copied into Windows:

x64	08/05/2025 1:47 pm	File folder	
DS_GUI.vcxproj	08/05/2025 3:25 pm	VC++ Project	7 KB
DS_GUI.vcxproj.filters	08/05/2025 3:03 pm	VC++ Project Filters F...	2 KB
DS_GUI.vcxproj.user	08/05/2025 1:47 pm	Per-User Project Opti...	1 KB
FILE1	11/05/2025 3:46 pm	Text Document	10,240 KB
<input checked="" type="checkbox"/> File3	11/05/2025 4:04 pm	File	3 KB
FileSystemManager.h	10/05/2025 11:09 pm	C/C++ Header	17 KB
helper_Function.h	10/05/2025 11:09 pm	C/C++ Header	11 KB
main.cpp	08/05/2025 3:31 pm	C++ Source	1 KB
MyForm.cpp	08/05/2025 1:50 pm	C++ Source	1 KB
MyForm.h	11/05/2025 3:46 pm	C/C++ Header	23 KB
MyForm.resx	10/05/2025 8:49 pm	Microsoft .NET Mana...	6 KB
Osblalh	08/05/2025 3:40 pm	File	1 KB

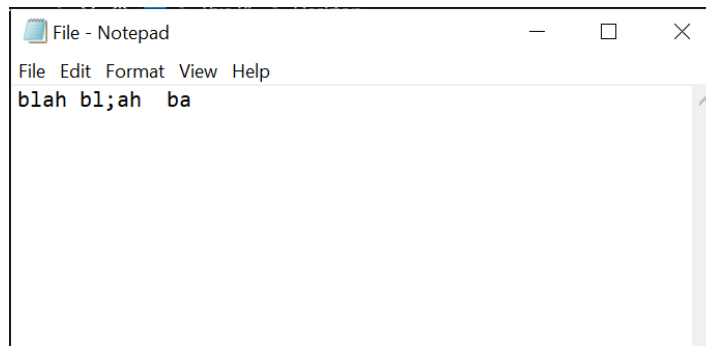
Content of file that got copied into Windows:



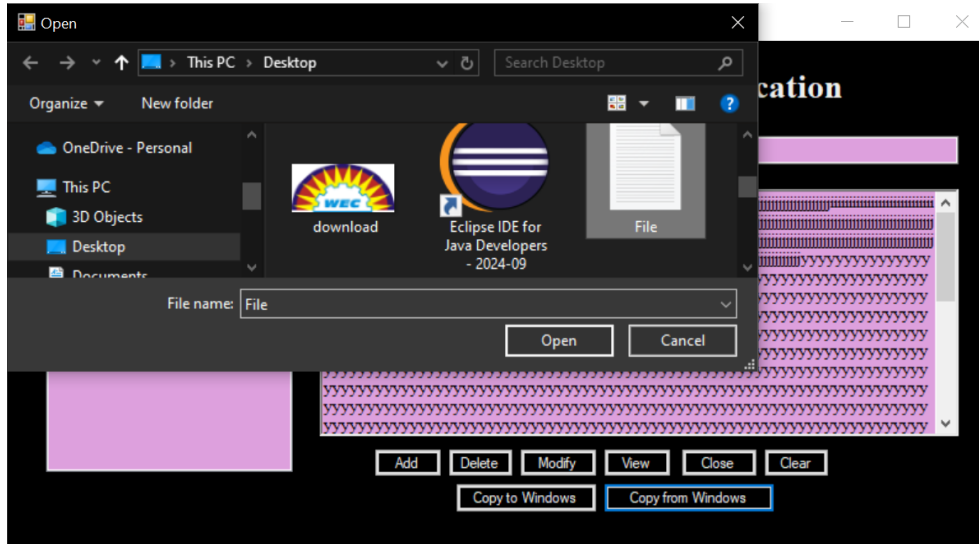
## 7. 7. Copying a File from Windows:

To import a file from the Windows system into the application, the user selects the Copy file from windows button. A file explorer opens from where the required file can be selected. The application reads its content and stores it internally using the file system's structure.

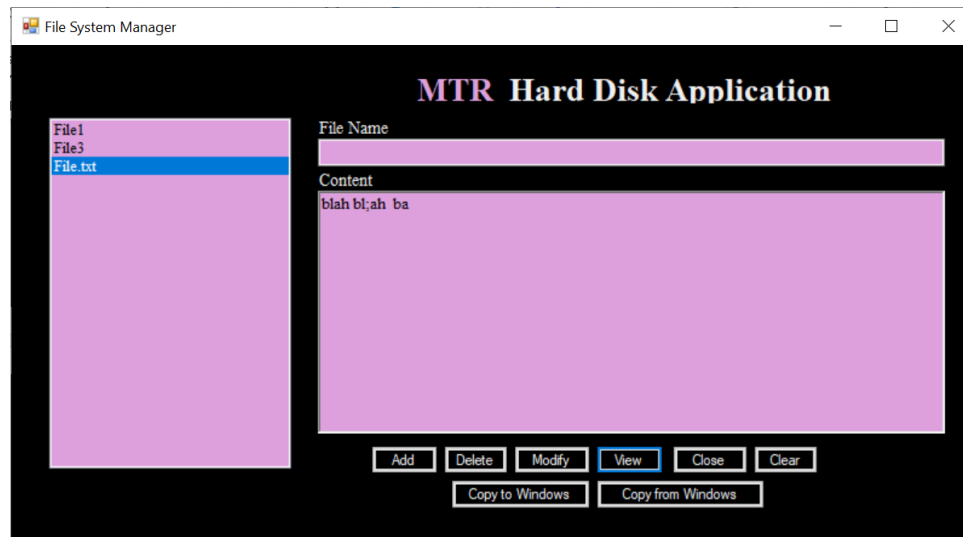
## The file to be copied



File explorer Window:

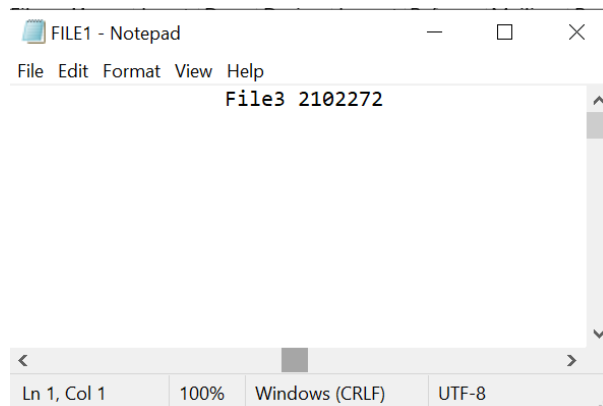
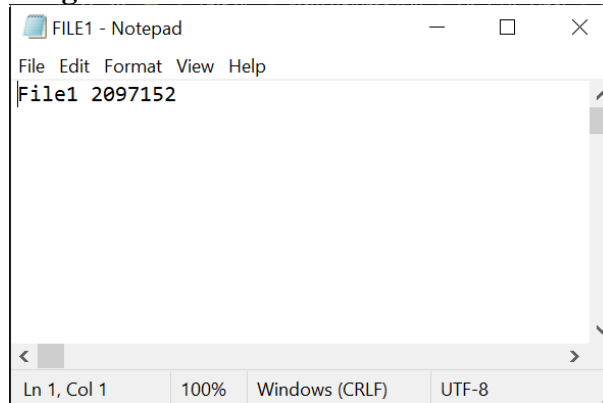


File copied:

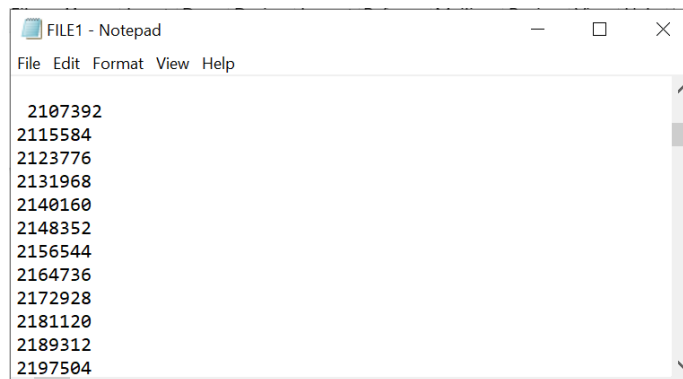


## **Results and Outputs:**

### **Filename area on files being added:**



### **Address minheap on three files of various length being added:**

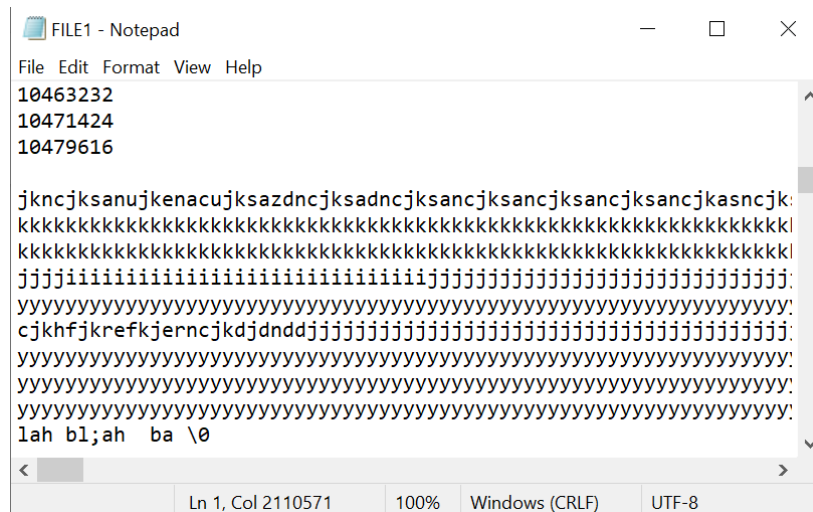


2107392  
2115584  
2123776  
2131968  
2140160  
2148352  
2156544  
2164736  
2172928  
2181120  
2189312  
2197504  
2205696  
2213888  
2222080  
2230272  
2238464  
2246656  
2254848  
2263040  
2271232  
2279424  
2287616  
2295808  
2304000  
2312192  
2320384  
2328576  
2336768  
2344960  
2353152  
2361344  
2369536  
2377728  
2385920  
2394112  
2402304  
2410496  
2418688  
2426880  
2435072  
2443264  
2451456

2108416  
2116608  
2124800  
2132992  
2141184  
2149376  
2157568  
2165760  
2173952  
2182144  
2190336  
2198528  
2206720  
2214912  
2223104  
2231296  
2239488  
2247680  
2255872  
2264064  
2272256  
2280448  
2288640  
2296832  
2305024  
2313216  
2321408  
2329600  
2337792  
2345984  
2354176  
2362368  
2370560  
2378752  
2386944  
2395136  
2403328  
2411520  
2419712  
2427904  
2436096  
2444288  
2452480

2109440  
2117632  
2125824  
2134016  
2142208  
2150400  
2158592  
2166784  
2174976  
2183168  
2191360  
2199552  
2207744  
2215936  
2224128  
2232320  
2240512  
2248704  
2256896  
2265088  
2273280  
2281472  
2289664  
2297856  
2306048  
2314240  
2322432  
2330624  
2338816  
2347008  
2355200  
2363392  
2371584  
2379776  
2387968  
2396160  
2404352  
2412544  
2420736  
2428928  
2437120  
2445312  
2453504

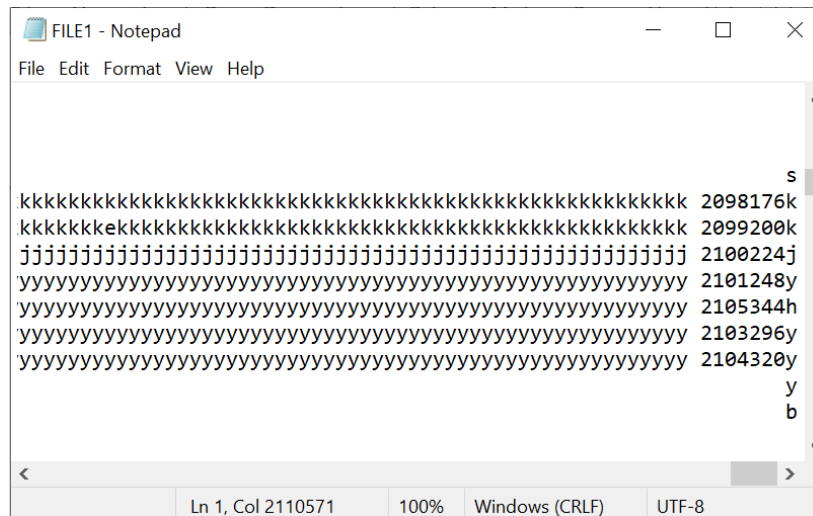
## FileContent Area having content of three files:



```
FILE1 - Notepad
File Edit Format View Help
10463232
10471424
10479616

jkncjksanujkenacujksazdncjksadncjksancjksancjksancjksancjksancjk:
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk|
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk|
jjjjiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiijjjjjjjjjjjjjjjjjjjjjjjjjjjjjj:
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy:
cjkfhfjkrfKjerncjkjdnddjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj:
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy:
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy:
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy:
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy:
lah bl;ah ba \0

Ln 1, Col 2110571 100% Windows (CRLF) UTF-8
```



```
FILE1 - Notepad
File Edit Format View Help

kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk s
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk 2098176k
jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj 2099200k
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy 2100224j
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy 2101248y
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy 2105344h
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy 2103296y
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy 2104320y
y
b

Ln 1, Col 2110571 100% Windows (CRLF) UTF-8
```

## **Challenged and Solutions:**

### **1. Efficient File Management with Minimal Time Complexity**

#### ***Challenge:***

One of the core challenges was selecting the appropriate data structures that ensured minimal time complexity for operations like file addition, deletion, modification, and retrieval. With frequent and dynamic file operations, maintaining performance and scalability became critical.

#### ***Solution:***

A hybrid approach was adopted combining **Vectors**, **Unordered Maps**, and **Linked Lists**.

- **Vectors** provided direct index access, reducing access time to  $O(1)$ .
- **Unordered Maps** were used to associate file names with vector indices, enabling fast lookup without sequential traversal.
- **Linked Lists** were used to store file contents block-by-block, supporting non-contiguous memory allocation and easy appending/modification.

This combination allowed efficient dynamic storage, fast access, and simplified block-level file handling.

### **2. Fragmentation and Memory Reuse**

#### ***Challenge:***

As files were added, modified, and deleted, memory fragmentation increased. Reusing freed blocks efficiently was necessary to avoid unnecessary memory expansion.

#### ***Solution:***

We used a **MinHeap** to track and sort available memory addresses.

- When a file was deleted, its blocks were added back to the heap.
- When adding new files, the heap provided the smallest available address, promoting efficient reuse and helping defragment over time.

### **3. Managing Non-Contiguous File Storage**

#### ***Challenge:***

File content often could not be stored in contiguous memory blocks due to dynamic insertions and deletions. This posed issues in organizing and retrieving data coherently.

#### ***Solution:***

**Linked Lists** were used to link blocks of a file, regardless of where in memory they were stored.

This structure:

- Preserved the logical sequence of file content,
- Supported appending new blocks during modification,
- Made traversal and viewing of full file content seamless.

## **Conclusion:**

This project successfully demonstrated the design and implementation of a simplified file system that efficiently manages file storage, access, and organization using a smart combination of data structures. By integrating **Vectors**, **Linked Lists**, **Unordered Maps**, and a **MinHeap**, the system achieved:

- **Fast and dynamic file access** with minimal time complexity,



- **Non-contiguous memory management** using linked blocks for file content,
- **Effective memory reuse** and automatic address sorting through a MinHeap,
- **Reliable file operations** such as creation, deletion, modification, and viewing,
- **Seamless integration** with external file systems for importing and exporting files.

These outcomes highlight not only the technical robustness of the solution but also its scalability and adaptability for more complex file management needs.

### **Potential for Further Development**

While the current version achieves core file system functionalities, several enhancements can be considered for future versions:

- **User authentication and file permissions** for multi-user support,
- **File categorization and directory trees** to support nested folders,
- **Search by file content or metadata**,
- **Improved GUI/UX** for better user interaction,
- **Automatic defragmentation algorithms** running in the background,
- **Persistent storage** using disk-based structures instead of runtime memory.

These improvements could elevate the project from a simulation to a practical mini file system with real-world applications.

### **External Tools:**

- YouTube
- Google
- AI Tools