3.1

To implement a Sequential File Allocation strategy in Python, we need to simulate the disk blocks and the file allocation table (FAT). Each student record will be stored as a file in contiguous blocks, and the FAT will keep track of the starting block and the length of each student record file.

**Python Program**

Here's a Python program that implements the Sequential File Allocation strategy for the school database:

```
class StudentRecord:
    def __init__(self, name, student_id, grade, address):
        self.name = name
        self.student_id = student_id
        self.grade = grade
        self.address = address
        self.size = 1  # Assuming each record takes 1 block

class FileAllocationTableEntry:
    def __init__(self, start_block, length):
        self.start_block = start_block
        self.length = length

class SequentialFileAllocation:
    def __init__(self, total_blocks):
        self.total_blocks = total_blocks
        self.disk = [None] * total_blocks  # Simulate disk blocks
        self.fat = {}  # File Allocation Table to store file info
        self.next_free_block = 0

    def add_record(self, student_record):
        if self.next_free_block + student_record.size > self.total_blocks:
            print("Not enough disk space to add record for student:", student_record.name)
            return False
```

```
        start_block = self.next_free_block
        # Store record in disk blocks
        for i in range(student_record.size):
            self.disk[start_block + i] = student_record

        # Update the FAT
        self.fat[student_record.student_id] = FileAllocationTableEntry(start_block, student_record.size)
        self.next_free_block += student_record.size
        return True

    def delete_record(self, student_id):
        if student_id not in self.fat:
            print("Record not found for student ID:", student_id)
            return False

        entry = self.fat.pop(student_id)
        # Clear the disk blocks
        for i in range(entry.start_block, entry.start_block + entry.length):
            self.disk[i] = None
        return True

    def update_record(self, student_id, name=None, grade=None, address=None):
        if student_id not in self.fat:
            print("Record not found for student ID:", student_id)
            return False

        entry = self.fat[student_id]
        # Get the student record
        student_record = self.disk[entry.start_block]
```

```
        # Update the student record fields
        if name:
            student_record.name = name
        if grade:
            student_record.grade = grade
        if address:
            student_record.address = address
        return True

    def display_records(self):
        print("File Allocation Table:")
        for student_id, entry in self.fat.items():
            student = self.disk[entry.start_block]
            print(f"ID: {student.student_id}, Name: {student.name}, Grade: {student.grade}, Address:
{student.address}")

        print("\nDisk Blocks:")
        for i, block in enumerate(self.disk):
            if block:
                print(f"Block {i}: {block.name} (ID: {block.student_id})")
            else:
                print(f"Block {i}: Free")

# Example usage:
sequential_file_allocation = SequentialFileAllocation(total_blocks=10)

# Add student records
sequential_file_allocation.add_record(StudentRecord("John Doe", 101, 10, "123 Main Street"))
sequential_file_allocation.add_record(StudentRecord("Jane Smith", 102, 11, "456 Elm Street"))
sequential_file_allocation.add_record(StudentRecord("Michael Brown", 103, 9, "789 Oak Avenue"))
```

```
# Display the records and disk usage
sequential_file_allocation.display_records()


# Update a student record
sequential_file_allocation.update_record(102, grade=12)


# Display the updated records
sequential_file_allocation.display_records()


# Delete a student record
sequential_file_allocation.delete_record(101)


# Display the records after deletion
sequential_file_allocation.display_records()
```

**Explanation of the Program**

1. **StudentRecord Class**: Represents a student record with fields for name, ID, grade, address, and a size (assumed to be 1 block per record).
2. **FileAllocationTableEntry Class**: Represents an entry in the File Allocation Table (FAT), which tracks the starting block and length of each file.
3. **SequentialFileAllocation Class**: Implements the sequential file allocation strategy.
    - **__init__**: Initializes the disk and FAT. The disk is a list simulating disk blocks, and fat is a dictionary that stores file allocation information.
    - **add_record**: Adds a new student record to the disk, updates the FAT, and adjusts the next free block.
    - **delete_record**: Deletes a student record by clearing the corresponding disk blocks and removing the entry from the FAT.
    - **update_record**: Updates fields of an existing student record.
    - **display_records**: Displays the contents of the FAT and the current state of the disk blocks.

**Disk Space and Storage Utilization**

- **Total Disk Space Used**: The program calculates this based on the number of blocks allocated for each student record.

- **Efficiency**: The efficiency is based on how well the disk space is utilized. In this simplified model, we assume each record uses one block, so the utilization is quite straightforward. In more complex scenarios, fragmentation and block size considerations would be critical.

This program provides a basic simulation of sequential file allocation for managing a student database in Python.

3.2

To implement an Indexed File Allocation strategy for a hospital's EMR system, we need to simulate disk blocks, index blocks, and an index table that maps each patient record to its respective index block. Each patient record will have its own index block containing pointers to the actual blocks of disk space where the record's data is stored.

**Python Program**

Here's a Python program that implements the Indexed File Allocation strategy for the hospital EMR system:

```
class PatientRecord:
    def __init__(self, name, age, medical_id, address):
        self.name = name
        self.age = age
        self.medical_id = medical_id
        self.address = address
        self.size = 1  # Assuming each record takes 1 block


class IndexBlock:
    def __init__(self):
        self.blocks = []  # List to store pointers to the actual data blocks


class IndexedFileAllocation:
    def __init__(self, total_blocks):
        self.total_blocks = total_blocks
        self.disk = [None] * total_blocks  # Simulate disk blocks
        self.index_table = {}  # Index table to map Medical ID to index block
        self.free_blocks = set(range(total_blocks))  # Set of free disk blocks
```

```python
def allocate_blocks(self, num_blocks):
    """Allocate 'num_blocks' from free blocks and return their indices."""
    if len(self.free_blocks) < num_blocks:
        return None  # Not enough space
    allocated = set()
    while len(allocated) < num_blocks:
        block = self.free_blocks.pop()
        allocated.add(block)
    return allocated

def add_record(self, patient_record):
    if patient_record.medical_id in self.index_table:
        print(f"Record for Medical ID {patient_record.medical_id} already exists.")
        return False

    index_block = IndexBlock()
    # Allocate blocks for the patient record
    allocated_blocks = self.allocate_blocks(patient_record.size)
    if not allocated_blocks:
        print("Not enough disk space to add record for patient:", patient_record.name)
        return False

    # Store pointers in index block
    index_block.blocks.extend(allocated_blocks)

    # Store the record in allocated disk blocks
    for block in allocated_blocks:
        self.disk[block] = patient_record

    # Update the index table
    self.index_table[patient_record.medical_id] = index_block
```

```
        return True

    def delete_record(self, medical_id):
        if medical_id not in self.index_table:
            print(f"Record not found for Medical ID: {medical_id}")
            return False

        index_block = self.index_table.pop(medical_id)
        # Free up the blocks
        for block in index_block.blocks:
            self.disk[block] = None
            self.free_blocks.add(block)
        return True

    def update_record(self, medical_id, name=None, age=None, address=None):
        if medical_id not in self.index_table:
            print(f"Record not found for Medical ID: {medical_id}")
            return False

        index_block = self.index_table[medical_id]
        # Retrieve the patient record from the disk
        patient_record = self.disk[index_block.blocks[0]]

        # Update the patient record fields
        if name:
            patient_record.name = name
        if age:
            patient_record.age = age
        if address:
            patient_record.address = address
        return True
```

```
    def retrieve_record(self, medical_id):
        if medical_id not in self.index_table:
            print(f"Record not found for Medical ID: {medical_id}")
            return None

        index_block = self.index_table[medical_id]
        # Retrieve the patient record from the first block
        patient_record = self.disk[index_block.blocks[0]]
        return patient_record

    def display_records(self):
        print("Index Table:")
        for medical_id, index_block in self.index_table.items():
            patient = self.disk[index_block.blocks[0]]
            print(f"Medical ID: {patient.medical_id}, Name: {patient.name}, Age: {patient.age}, Address: {patient.address}")

        print("\nDisk Blocks:")
        for i, block in enumerate(self.disk):
            if block:
                print(f"Block {i}: {block.name} (Medical ID: {block.medical_id})")
            else:
                print(f"Block {i}: Free")

# Example usage:
indexed_file_allocation = IndexedFileAllocation(total_blocks=10)

# Add patient records
indexed_file_allocation.add_record(PatientRecord("John Smith", 45, 1001, "123 Hospital Road"))
indexed_file_allocation.add_record(PatientRecord("Jane Doe", 32, 1002, "456 Clinic Avenue"))
```

indexed_file_allocation.add_record(PatientRecord("Michael Johnson", 58, 1003, "789 Medical Plaza"))

# Display the records and disk usage

indexed_file_allocation.display_records()

# Update a patient record

indexed_file_allocation.update_record(1002, age=33)

# Display the updated records

indexed_file_allocation.display_records()

# Delete a patient record

indexed_file_allocation.delete_record(1001)

# Display the records after deletion

indexed_file_allocation.display_records()

**Explanation of the Program**

1. **PatientRecord Class**: Represents a patient record with fields for name, age, medical ID, address, and a size (assumed to be 1 block per record).
2. **IndexBlock Class**: Represents an index block containing pointers to the actual data blocks that store a patient's record.
3. **IndexedFileAllocation Class**: Implements the Indexed File Allocation strategy.
    - **__init__**: Initializes the disk, the index table, and the set of free blocks. The disk simulates the disk blocks, index_table maps medical IDs to index blocks, and free_blocks tracks available disk blocks.
    - **allocate_blocks**: Allocates a specified number of disk blocks from the pool of free blocks.
    - **add_record**: Adds a new patient record to the disk, allocates necessary blocks, and updates the index table.
    - **delete_record**: Deletes a patient record by freeing the associated disk blocks and removing the entry from the index table.
    - **update_record**: Updates fields of an existing patient record.

- **retrieve_record**: Retrieves a patient record based on the medical ID.
- **display_records**: Displays the contents of the index table and the current state of the disk blocks.

**Disk Space and Storage Utilization**

- **Total Disk Space Used**: This is determined by the number of blocks allocated for each patient record and their corresponding index blocks.
- **Efficiency**: The Indexed File Allocation strategy is efficient in terms of access times since it uses direct indexing to access records. It also handles deletions and updates more gracefully compared to sequential allocation, reducing fragmentation.

This program provides a basic simulation of the Indexed File Allocation strategy for managing a hospital's EMR system in Python.

3.3

To implement the Linked File Allocation strategy for a multimedia application in Python, we need to simulate the storage of digital media files in non-contiguous blocks of disk space, where each block points to the next block of the file. We will also maintain a File Allocation Table (FAT) to keep track of the starting block of each file.

**Python Program**

Here's a Python program that implements the Linked File Allocation strategy for managing digital media files:

```
class MediaFile:
    def __init__(self, file_name, file_type, size):
        self.file_name = file_name
        self.file_type = file_type
        self.size = size  # Size in MB
        self.blocks = []  # List to store blocks allocated for this file


class DiskBlock:
    def __init__(self):
        self.data = None  # Placeholder for media file data
        self.next_block = None  # Pointer to the next block


class LinkedFileAllocation:
    def __init__(self, block_size, total_blocks):
        self.block_size = block_size  # Size of each block in MB
```

```
        self.total_blocks = total_blocks
        self.disk = [DiskBlock() for _ in range(total_blocks)]  # Simulate disk blocks
        self.fat = {}  # File Allocation Table (FAT) to store file info
        self.free_blocks = set(range(total_blocks))  # Set of free disk blocks

    def allocate_blocks(self, file_size):
        """Allocate the required number of blocks for the file based on its size."""
        num_blocks = (file_size + self.block_size - 1) // self.block_size  # Calculate number of blocks
needed
        if len(self.free_blocks) < num_blocks:
            return None  # Not enough space

        allocated_blocks = []
        for _ in range(num_blocks):
            block = self.free_blocks.pop()
            allocated_blocks.append(block)
        return allocated_blocks

    def add_file(self, media_file):
        if media_file.file_name in self.fat:
            print(f"File {media_file.file_name} already exists.")
            return False

        allocated_blocks = self.allocate_blocks(media_file.size)
        if not allocated_blocks:
            print(f"Not enough disk space to add file: {media_file.file_name}")
            return False

        # Store the file in allocated disk blocks
        for i in range(len(allocated_blocks)):
            block_index = allocated_blocks[i]
```

```
            self.disk[block_index].data = media_file
            if i < len(allocated_blocks) - 1:
                self.disk[block_index].next_block = allocated_blocks[i + 1]

        # Update the file's block list and FAT
        media_file.blocks = allocated_blocks
        self.fat[media_file.file_name] = allocated_blocks[0]  # Store the starting block in FAT
        return True

    def delete_file(self, file_name):
        if file_name not in self.fat:
            print(f"File {file_name} not found.")
            return False

        # Get the starting block from the FAT
        current_block = self.fat.pop(file_name)

        # Traverse through the linked blocks and free them
        while current_block is not None:
            next_block = self.disk[current_block].next_block
            self.disk[current_block] = DiskBlock()  # Reset the block
            self.free_blocks.add(current_block)
            current_block = next_block
        return True

    def retrieve_file(self, file_name):
        if file_name not in self.fat:
            print(f"File {file_name} not found.")
            return None

        # Get the starting block from the FAT
```

```
        current_block = self.fat[file_name]
        file_data = []

        # Traverse through the linked blocks to gather the file data
        while current_block is not None:
            file_data.append(self.disk[current_block].data)
            current_block = self.disk[current_block].next_block
        return file_data[0]  # Return the file data (the MediaFile object)

    def display_disk_usage(self):
        print("File Allocation Table (FAT):")
        for file_name, start_block in self.fat.items():
            media_file = self.disk[start_block].data
            print(f"File: {file_name}, Type: {media_file.file_type}, Size: {media_file.size} MB, Starting Block:
{start_block}")

        print("\nDisk Blocks:")
        for i, block in enumerate(self.disk):
            if block.data:
                print(f"Block {i}: {block.data.file_name} -> Next Block: {block.next_block}")
            else:
                print(f"Block {i}: Free")

# Example usage:
linked_file_allocation = LinkedFileAllocation(block_size=5, total_blocks=20)

# Add media files
linked_file_allocation.add_file(MediaFile("Landscape.jpg", "Image", 5))
linked_file_allocation.add_file(MediaFile("Concert.mp4", "Video", 50))
linked_file_allocation.add_file(MediaFile("Song.mp3", "Audio", 8))
```

```
# Display the disk usage and FAT
linked_file_allocation.display_disk_usage()


# Retrieve a file
retrieved_file = linked_file_allocation.retrieve_file("Concert.mp4")
if retrieved_file:
    print(f"\nRetrieved File: {retrieved_file.file_name}, Type: {retrieved_file.file_type}, Size: {retrieved_file.size} MB")


# Delete a file
linked_file_allocation.delete_file("Landscape.jpg")


# Display the disk usage after deletion
linked_file_allocation.display_disk_usage()
```

**Explanation of the Program**

1. **MediaFile Class**: Represents a digital media file with fields for file name, type, size, and blocks allocated.
2. **DiskBlock Class**: Represents a disk block, which can store data (a media file) and a pointer (next_block) to the next block.
3. **LinkedFileAllocation Class**: Implements the Linked File Allocation strategy.
    - **__init__**: Initializes the disk blocks, the FAT, and the set of free blocks. The disk simulates the disk blocks, fat maps file names to the starting block, and free_blocks tracks available disk blocks.
    - **allocate_blocks**: Allocates a specified number of disk blocks based on the file size.
    - **add_file**: Adds a new file to the disk, allocates necessary blocks, and updates the FAT.
    - **delete_file**: Deletes a file by freeing the associated disk blocks and removing the entry from the FAT.
    - **retrieve_file**: Retrieves a file based on the file name by traversing through the linked blocks.
    - **display_disk_usage**: Displays the contents of the FAT and the current state of the disk blocks.

**Disk Space and Storage Utilization**

- **Total Disk Space Used**: This is determined by the number of blocks allocated for each file. The program calculates this based on the file size and block size.
- **Efficiency**: The Linked File Allocation strategy is efficient in terms of handling fragmentation, as it does not require contiguous blocks. However, access times can be slower if the file is spread out over many non-contiguous blocks.

This program provides a basic simulation of the Linked File Allocation strategy for managing digital media files in a multimedia application in Python.

3.4

To implement the Sequential File Allocation strategy for the "DigitalArchive" storage system, we will create a Python program that simulates the storage of digital image files for a museum's photograph collection. The program will manage the allocation of disk space, handle new file insertions, and display the allocation status.

```
class Photograph:
    def __init__(self, file_name, size):
        self.file_name = file_name
        self.size = size  # Size in MB
        self.blocks = []  # List of blocks allocated for this file



class SequentialFileAllocation:
    def __init__(self, block_size, total_blocks):
        self.block_size = block_size  # Size of each block in MB
        self.total_blocks = total_blocks
        self.disk = [None] * total_blocks  # Simulate disk blocks with None (unallocated)
        self.fat = {}  # File Allocation Table (FAT) to store file information

    def allocate_blocks(self, file_size):
        """Allocate contiguous blocks for a file based on its size."""
        num_blocks_needed = (file_size + self.block_size - 1) // self.block_size
        start_index = self.find_contiguous_blocks(num_blocks_needed)

        if start_index == -1:
```

```
            return None  # Not enough contiguous space

        # Allocate the blocks
        for i in range(start_index, start_index + num_blocks_needed):
            self.disk[i] = True  # Mark the block as allocated

        return list(range(start_index, start_index + num_blocks_needed))

    def find_contiguous_blocks(self, num_blocks_needed):
        """Find a sequence of contiguous free blocks."""
        count = 0
        start_index = -1

        for i in range(self.total_blocks):
            if self.disk[i] is None:  # Block is free
                if count == 0:
                    start_index = i
                count += 1
                if count == num_blocks_needed:
                    return start_index
            else:
                count = 0  # Reset count if a block is not free

        return -1  # No sufficient contiguous blocks found

    def add_file(self, photograph):
        if photograph.file_name in self.fat:
            print(f"File {photograph.file_name} already exists.")
            return False

        allocated_blocks = self.allocate_blocks(photograph.size)
```

```
        if not allocated_blocks:
            print(f"Not enough disk space to add file: {photograph.file_name}")
            return False

        photograph.blocks = allocated_blocks
        self.fat[photograph.file_name] = photograph
        print(f"File {photograph.file_name} added successfully.")
        return True

    def delete_file(self, file_name):
        if file_name not in self.fat:
            print(f"File {file_name} not found.")
            return False

        photograph = self.fat.pop(file_name)

        # Free the allocated blocks
        for block in photograph.blocks:
            self.disk[block] = None

        print(f"File {file_name} deleted successfully.")
        return True

    def display_disk_usage(self):
        print("\nFile Allocation Table (FAT):")
        for file_name, photograph in self.fat.items():
            print(f"File: {file_name}, Size: {photograph.size} MB, Blocks: {photograph.blocks}")

        print("\nDisk Blocks:")
        for i in range(self.total_blocks):
            if self.disk[i] is None:
```

```
                print(f"Block {i}: Free")
            else:
                print(f"Block {i}: Allocated")

    def calculate_total_disk_space_used(self):
        used_blocks = sum(1 for block in self.disk if block is not None)
        total_space_used = used_blocks * self.block_size
        print(f"\nTotal Disk Space Used: {total_space_used} MB")
        return total_space_used


# Example usage
sequential_allocation = SequentialFileAllocation(block_size=10, total_blocks=100)

# Add photographs
sequential_allocation.add_file(Photograph("portrait1.jpg", 15))
sequential_allocation.add_file(Photograph("landscape1.jpg", 25))
sequential_allocation.add_file(Photograph("architecture1.jpg", 5))
sequential_allocation.add_file(Photograph("portrait2.jpg", 12))

# Display the disk usage and FAT
sequential_allocation.display_disk_usage()

# Calculate total disk space used
sequential_allocation.calculate_total_disk_space_used()

# Delete a file
sequential_allocation.delete_file("portrait1.jpg")

# Display the disk usage after deletion
sequential_allocation.display_disk_usage()
```

# Calculate total disk space used after deletion

sequential_allocation.calculate_total_disk_space_used()

# Try adding a new photograph after deletion

sequential_allocation.add_file(Photograph("new_photo.jpg", 10))

# Display the disk usage after adding a new file

sequential_allocation.display_disk_usage()

**Explanation of the Program**

1. **Photograph Class**: Represents a photograph with attributes like file name, size (in MB), and the list of blocks allocated for this file.
2. **SequentialFileAllocation Class**: Manages the sequential allocation of disk blocks for storing photographs.
   - **__init__**: Initializes the disk blocks, FAT, and other necessary attributes.
   - **allocate_blocks**: Allocates contiguous blocks for a file based on its size.
   - **find_contiguous_blocks**: Finds a sequence of contiguous free blocks required to store a file.
   - **add_file**: Adds a new file by allocating the necessary blocks and updates the FAT.
   - **delete_file**: Deletes a file by freeing its allocated blocks and removing its entry from the FAT.
   - **display_disk_usage**: Displays the FAT and the current state of disk blocks, showing which blocks are allocated and which are free.
   - **calculate_total_disk_space_used**: Calculates and displays the total disk space used by summing up the allocated blocks.

**Key Features of the Sequential Allocation Strategy**

- **Contiguous Allocation**: Files are stored in contiguous blocks, which provides good performance for sequential access since there is no need to follow pointers between non-contiguous blocks.
- **Disk Space Utilization**: It effectively uses disk space as long as there is a sufficient number of contiguous free blocks. However, fragmentation can become an issue over time as files are added and deleted.
- **Efficiency**: Sequential allocation is efficient for reading and writing large files because all data is stored in contiguous blocks.

This program simulates the management of digital photographs using a sequential file allocation strategy, demonstrating how files are stored, retrieved, and deleted while maintaining efficient disk space utilization.

3.5

To implement the Index Sequential File Allocation strategy for OS EnterpriseX, we need to create a Python program that simulates the management and storage of large files, such as databases and multimedia files. This program will use an index structure (e.g., a dictionary) to manage the allocation of disk blocks and allow efficient data retrieval.

Here's a Python program to simulate this file allocation strategy:

```
class File:
    def __init__(self, file_name, file_type, size):
        self.file_name = file_name
        self.file_type = file_type
        self.size = size  # Size in KB
        self.blocks = []  # List of blocks allocated to this file


class IndexSequentialAllocation:
    def __init__(self, block_size, total_blocks):
        self.block_size = block_size  # Size of each block in KB
        self.total_blocks = total_blocks
        self.disk = [None] * total_blocks  # Simulate disk blocks with None (unallocated)
        self.index = {}  # Index structure (dictionary) to store file and block mapping

    def allocate_blocks(self, file_size):
        """Allocate sequential blocks for a file based on its size."""
        num_blocks_needed = (file_size + self.block_size - 1) // self.block_size
        allocated_blocks = []

        for i in range(self.total_blocks):
            # Check if there are enough consecutive free blocks
            if self.disk[i] is None and all(self.disk[i + j] is None for j in range(num_blocks_needed)):
```

```
            allocated_blocks = list(range(i, i + num_blocks_needed))
            break

    if not allocated_blocks:
        return None  # Not enough contiguous space

    # Allocate the blocks
    for block in allocated_blocks:
        self.disk[block] = True  # Mark block as allocated

    return allocated_blocks

def add_file(self, file):
    if file.file_name in self.index:
        print(f"File {file.file_name} already exists.")
        return False

    allocated_blocks = self.allocate_blocks(file.size)
    if not allocated_blocks:
        print(f"Not enough disk space to add file: {file.file_name}")
        return False

    file.blocks = allocated_blocks
    self.index[file.file_name] = file
    print(f"File {file.file_name} added successfully with blocks {allocated_blocks}.")
    return True

def delete_file(self, file_name):
    if file_name not in self.index:
        print(f"File {file_name} not found.")
        return False
```

```
        file = self.index.pop(file_name)

        # Free the allocated blocks
        for block in file.blocks:
            self.disk[block] = None

        print(f"File {file_name} deleted successfully.")
        return True

    def display_disk_usage(self):
        print("\nIndex Structure (File to Blocks Mapping):")
        for file_name, file in self.index.items():
            print(f"File: {file_name}, Type: {file.file_type}, Size: {file.size} KB, Blocks: {file.blocks}")

        print("\nDisk Blocks:")
        for i in range(self.total_blocks):
            if self.disk[i] is None:
                print(f"Block {i}: Free")
            else:
                print(f"Block {i}: Allocated")

    def retrieve_file_blocks(self, file_name):
        """Retrieve the blocks allocated to a specific file."""
        if file_name not in self.index:
            print(f"File {file_name} not found.")
            return None
        file = self.index[file_name]
        print(f"Blocks for file {file_name}: {file.blocks}")
        return file.blocks
```

```
    def calculate_total_disk_space_used(self):
        used_blocks = sum(1 for block in self.disk if block is not None)
        total_space_used = used_blocks * self.block_size
        print(f"\nTotal Disk Space Used: {total_space_used} KB")
        return total_space_used


# Example usage
index_allocation = IndexSequentialAllocation(block_size=8, total_blocks=1000)

# Add files
index_allocation.add_file(File("database1.db", "database", 320))
index_allocation.add_file(File("video1.mp4", "multimedia", 100))
index_allocation.add_file(File("image1.jpg", "multimedia", 16))
index_allocation.add_file(File("document1.txt", "text", 4))

# Display the disk usage and index structure
index_allocation.display_disk_usage()

# Calculate total disk space used
index_allocation.calculate_total_disk_space_used()

# Retrieve blocks of a file
index_allocation.retrieve_file_blocks("database1.db")

# Delete a file
index_allocation.delete_file("video1.mp4")

# Display the disk usage after deletion
index_allocation.display_disk_usage()
```

```
# Calculate total disk space used after deletion
index_allocation.calculate_total_disk_space_used()


# Try adding a new file after deletion
index_allocation.add_file(File("new_video.mp4", "multimedia", 200))


# Display the disk usage after adding a new file
index_allocation.display_disk_usage()
```

**Explanation of the Program**

1. **File Class**: Represents a file with attributes like file name, type, size (in KB), and the list of blocks allocated for this file.
2. **IndexSequentialAllocation Class**: Manages the index sequential allocation of disk blocks for storing files.
   - **__init__**: Initializes the disk blocks, index structure, and other necessary attributes.
   - **allocate_blocks**: Allocates sequential blocks for a file based on its size.
   - **add_file**: Adds a new file by allocating the necessary blocks and updates the index structure.
   - **delete_file**: Deletes a file by freeing its allocated blocks and removing its entry from the index.
   - **display_disk_usage**: Displays the index structure and the current state of disk blocks, showing which blocks are allocated and which are free.
   - **retrieve_file_blocks**: Retrieves and displays the blocks allocated to a specific file based on its name.
   - **calculate_total_disk_space_used**: Calculates and displays the total disk space used by summing up the allocated blocks.

**Key Features of the Index Sequential Allocation Strategy**

- **Index Structure**: The program uses an index structure (dictionary) to map files to their respective blocks, allowing quick access to specific data blocks.
- **Sequential Storage**: Files are stored sequentially in terms of logical organization, but the physical storage is optimized using the index structure, which reduces fragmentation and improves access times.
- **Efficiency**: Index sequential allocation is efficient for both sequential and random access, making it suitable for large-scale data processing and storage applications like OS EnterpriseX.

- **Fragmentation Management**: By maintaining an index structure, the program can manage fragmentation effectively, as blocks are allocated sequentially and can be easily rearranged if needed.

This program simulates the management of large files using an index sequential file allocation strategy, demonstrating how files are stored, retrieved, and deleted while maintaining efficient disk space utilization and quick data access.