

**SVKM’s NMIMS**

**Mukesh Patel School of Technology Management and Engineering**

**Vile Parle West Mumbai-56**

**B Tech Computer Science and Engineering (Data Science) -311 Program**

**Semester – IV**

**Academic Year 2023-24**

**Design and Analysis of Algorithms**

**Project Report**

Prepared by

**Gautami Ankam 001**

**Anuj Iyer 011**

**Siya Jain 012**

**Soham Joshi 013**

**Faculty In charge**

Prof. Sofia Francis



**SVKM’s NMIMS**

**Mukesh Patel School of Technology Management and Engineering**

**Vile Parle West Mumbai-56**

**Index**

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Topics** | **Page Numbers** |
| **1** | **Introduction**   1. Digital Overload and Photo Management 2. Project Objectives 3. System Overview |  |
| **2** | **Architecture**   1. Core System (C++)    1. Photo Storage & Retrieval    2. Metadata Indexing    3. Sorting Mechanisms    4. Recent Photos Tracking    5. Searching Algorithms |  |
| **3** | **Problem Statement**   1. Challenges in Traditional Systems 2. Project Goals |  |
| **4** | **Scope**   1. Photo Storage & Retrieval 2. Metadata-Based Search & Filtering 3. Sorting Mechanisms 4. Recent Photos Tracking |  |
| **5** | **Societal Need**   1. Digital Overload 2. User Experience 3. Accessibility 4. Privacy and Security |  |
| **6** | **List Of Functions**   1. PhotoGallerySystem class functions 2. Helper Functions |  |
| **7** | **Algorithms Used**   1. Quicksort 2. Binary Search 3. KMP Algorithm (Backtracking) |  |
| **8** | **Data Structures**   1. AVL Tree 2. Trie 3. Priority Queue (Max Heap) 4. Hash Map 5. Linked List |  |
| **9** | **Time Complexity** |  |
| **10** | **Code** |  |
| **11** | **Results** |  |
| **12** | **Future Enhancements**   1. Cloud Integration 2. Enhanced AI Models 3. User Collaboration 4. Mobile Compatibility 5. Voice Control 6. Augmented Reality (AR) |  |
| **13** | **Conclusion**   1. Project Summary 2. Future Work |  |

**Introduction**

In the digital age, the volume of digital photos has surged exponentially, driven by the widespread adoption of smartphones, digital cameras, and cloud storage solutions. As individuals capture countless moments daily, managing these vast and ever-growing collections of images has become a significant challenge. Traditional photo management systems, though functional at a basic level, often fall short when it comes to offering intuitive and scalable solutions for organizing, searching, and retrieving digital photographs. Users frequently face frustration when attempting to locate specific photos based on various criteria such as date, location, event, or manually assigned tags. The lack of dynamic search capabilities, smart categorization, and context-aware filtering makes these systems inefficient, especially as photo libraries expand into the tens of thousands of images.

Furthermore, many existing systems are limited to rudimentary functions like chronological sorting or folder-based organization. Advanced features that could greatly enhance user experience—such as flexible tagging systems, fast retrieval, and multi-criteria search—are either absent or not optimized for large-scale use. As a result, users are compelled to spend considerable time manually sorting through their collections or tagging images, leading to inefficiencies and reduced usability.

This project proposes to bridge this gap by designing and implementing a comprehensive and efficient Photo Gallery System tailored for the modern user. The system will be engineered to handle large volumes of image data while ensuring fast and accurate access to photos through a highly responsive and user-friendly interface. At its core, the system will utilize efficient, **custom-built data structures and algorithms in C++** to manage photo metadata, index images for quick lookup, and implement optimized sorting and search mechanisms based on attributes such as timestamp, geolocation, tags, and file properties.

The system will incorporate five key data structures—**AVL Tree, Trie, Priority Queue, Hash Map, and Linked List**—to support a wide range of core operations, including balanced storage, keyword-based indexing, prioritized photo viewing, fast lookup, and chronological organization. Additionally, it will feature classic algorithms such as **QuickSort for efficient ordering, Binary Search for rapid retrieval, and KMP String Matching for advanced keyword searches**.

Moreover, the system will maintain a history of recently viewed or uploaded photos, support customizable albums, and provide basic insights into photo usage and access patterns. The use of **SQLite as the backend database** ensures lightweight, fast, and persistent storage of metadata without introducing external dependencies.

**Project Objectives**

The primary objective of this project is to design and implement a **robust, intelligent, and scalable Photo Gallery System** that enables users to efficiently store, search, organize, and manage large collections of digital photographs. With the exponential growth in image data, the system aims to overcome the limitations of conventional photo management applications by integrating advanced data structures, efficient algorithms, and AI-based functionalities.

Key objectives include:

* **Efficient Photo Storage and Retrieval:** Implement a structured approach to store photos along with rich metadata such as filename, date, location, and user-defined tags. Retrieval operations should be swift and scalable even for large datasets.
* **Advanced Metadata-Based Search and Sorting:** Enable users to search and filter images using multiple criteria like date, location, and tags with minimal latency through optimized searching and sorting mechanisms.
* **Recent Photo Tracking:** Keep track of recently added or viewed photos using data structures like priority queues to enhance user experience.
* **User-Friendly Interface:** Offer an intuitive and interactive user interface that supports seamless navigation, searching, sorting,.

**Core Components and Features:**

* **Photo Storage & Metadata Indexing:**
  + Photos are loaded from a file system or database into a std::vector.
  + Metadata such as date, location, tags, and descriptions are indexed using std::unordered\_map for fast lookups.
* **Metadata-Based Search & Filtering:**
  + Implements keyword and tag-based searches using a **Trie**.
  + Supports location and date-based searches with efficient use of hash maps and **binary search** on sorted structures.
* **Sorting and Retrieval:**
  + Uses **Quick Sort** or **Merge Sort** to organize images based on metadata like date, size, or user-defined popularity.
  + Retrieves images through optimized data structures like vectors and AVL trees for range-based queries.
* **Recent Photo Tracking:**
  + Maintains a **priority queue (heap)** to track the most recently added or frequently viewed photos.

**System Architecture & Workflow:**

1. **Loading Phase:**
   * Load image data and metadata into appropriate C++ containers.
   * Build indexes for fast search and retrieval.
2. **User Interaction Layer:**
   * A clean, interactive UI built in C++ displays photos and offers options for search, filtering, sorting, and viewing.
3. **Output and Visualization:**
   * Results from AI processing are relayed back to the UI, updating the metadata, and categorization accordingly.

**Problem Statement**

In the digital era, managing large collections of photos efficiently is a crucial challenge. Traditional photo management systems lack efficient searching, sorting, and categorization features. Users often struggle to find specific photos based on metadata such as date, location, or tags. Additionally, advanced functionalities like automatic tagging, facial recognition, and content-based image search are missing in many conventional systems.

The goal of this project is to design and implement a Photo Gallery System that enables users to efficiently store, search, and manage their photos using optimized data structures and algorithms. The system will include functionalities such as metadata-based search, sorting, recent photo tracking, and advanced AI-based features like automatic tagging, face recognition, and similarity search.

**1. Challenges in Traditional Systems**

* Lack of efficient searching, sorting, and categorization features.
* Difficulty in locating specific photos using metadata like date, location, or tags.
* Poor scalability with large photo collections.
* Limited support for intelligent or AI-assisted photo management.

**2. Project Goals**

* Design and develop a **Photo Gallery System** that overcomes limitations of traditional systems.
* Implement optimized **data structures and algorithms** for fast and efficient photo storage, search, and management.
* Include key functionalities such as:
  + Metadata-based search and filtering
  + Efficient sorting mechanisms
  + Tracking of recently viewed or added photos

**Scope**

The scope of this project is to develop a comprehensive Photo Gallery System that addresses the limitations of traditional photo management systems. The system will provide the following key functionalities:

1. Photo Storage & Retrieval: Efficient storage and retrieval of photos along with their metadata.
2. Metadata-Based Search & Filtering: Fast search capabilities based on location, date, tags, and descriptions.
3. Sorting Mechanisms: Efficient sorting of photos based on date, size, and popularity.
4. Recent Photos Tracking: Maintaining a priority queue of the most recently added or viewed photos.

**Societal Need**

**1. Digital Overload**

With the rise of smartphones and digital cameras, users are capturing thousands of photos each year. However, traditional photo management systems often become cluttered and inefficient when dealing with such large volumes. The proposed Photo Gallery System tackles this "digital overload" by offering smart indexing, filtering, and organization—helping users avoid endless scrolling or manual folder sorting.

**2. User Experience**

Modern users expect fast and intuitive interfaces. This system improves user experience by offering instant search results, auto-created albums, and smart tagging. Features like recent photo tracking and advanced sorting allow users to engage with their memories in a smoother and more meaningful way, reducing frustration and saving time.

**3. Accessibility**

Manually browsing large libraries can be overwhelming or difficult for some users. With intelligent search and automatic categorization, this system makes photo management more accessible for users of all ages and skill levels. Whether searching by voice (in the future) or simply viewing recent memories, accessibility is enhanced at every level.

**4. Privacy and Security**

Unlike many cloud-based services that store personal photos on remote servers, this system is designed with user privacy in mind. It can run locally, meaning users retain control over their data. Additionally, future versions could offer encryption for metadata and images, ensuring sensitive content is protected from unauthorized access.

**Architecture**

**System Architecture**

* **Photo Storage & Retrieval:**  
  Photos are stored in a std::vector along with metadata (filename, location, date, tags). This allows easy and quick access.
* **Metadata Indexing:**  
  A std::unordered\_map is used to index metadata like tags or locations for fast searching.
* **Sorting Mechanisms:**  
  Quick Sort or Merge Sort is used to sort photos by date, size, or popularity. Sorted data is also stored in trees like BST or AVL Tree for efficient range searches.
* **Recent Photos Tracking:**  
  A priority queue (heap) keeps track of the most recently viewed or added photos for quick access.
* **Searching Algorithms:** Searching by date, location, and tags implemented by various algorithms and data structures.

### **List of Functions**

PhotoGallerySystem **class functions**:

* addPhoto(): Adds a new photo to the gallery.
* viewPhoto(): Increments the view count of a specific photo.
* deletePhoto(): Deletes a photo from the gallery.
* searchByLocation(): Searches for photos by location.
* searchByTag(): Searches for photos by tag.
* searchByDateRange(): Searches for photos within a date range.
* searchByPrefix(): Searches for photos by keyword prefix using a Trie.
* searchByDescription(): Searches for photos by description text using the KMP algorithm.
* sortByDate(): Sorts photos by date.
* sortBySize(): Sorts photos by file size.
* sortByPopularity(): Sorts photos by view count (popularity).
* getMostRecentPhotos(): Retrieves the most recent photos.
* getMostPopularPhotos(): Retrieves the most popular photos.
* displayPhoto(): Displays details of a specific photo.
* displayAllPhotos(): Displays all photos in the gallery.
* getDataStructureStats(): Displays statistics about the data structures use

2. **Helper functions**:

* stringToTime(): Converts a date string to a time\_t object.
* timeToString(): Converts a time\_t object to a date string.
* partitionArray(): Partitions an array for the QuickSort algorithm.
* quickSort(): Sorts an array using the QuickSort algorithm.
* binarySearchDate(): Performs a binary search for a date in a sorted array.
* computeLPSArray(): Computes the Longest Prefix Suffix (LPS) array for the KMP algorithm.
* KMPSearch(): Searches for a pattern in a text using the KMP algorithm.

### **Algorithms Used**

**1. QuickSort Algorithm**:

* **Purpose**: To sort arrays of photos by different criteria (date, size, view count).
* **Steps**:
  + Choose a pivot element from the array.
  + Partition the array such that elements less than the pivot are on the left, and elements greater than the pivot are on the right.
  + Recursively apply the above steps to the sub-arrays formed by partitioning.
* **Sudo Code**:

function quickSort(arr, low, high):

if low < high:

pi = partitionArray(arr, low, high)

quickSort(arr, low, pi - 1)

quickSort(arr, pi + 1, high)

* **Time Complexity**: O(nlogn) on average, O(n2) in the worst case.
* **Space Complexity**: O(logn) due to recursion stack.

**2. Binary Search Algorithm**:

* **Purpose**: To search for a specific date in a sorted array of photos.
* **Steps**:

1. Compare the target date with the middle element of the array.
2. If the target date matches the middle element, return the index.
3. If the target date is less than the middle element, repeat the search on the left sub-array.
4. If the target date is greater than the middle element, repeat the search on the right sub-array.

* **Sudo Code**:

function binarySearchDate(arr, left, right, date):

if right >= left:

mid = left + (right - left) / 2

if arr[mid].date == date:

return mid

if arr[mid].date > date:

return binarySearchDate(arr, left, mid - 1, date)

return binarySearchDate(arr, mid + 1, right, date)

return -1

* **Time Complexity**: O(logn).
* **Space Complexity**: O(1).

**3. KMP (Knuth-Morris-Pratt) Algorithm**:

* **Purpose**: To search for a pattern (text) within a larger text (description).
* **Steps**:
  + 1. Preprocess the pattern to create an LPS (Longest Prefix Suffix) array.
    2. Use the LPS array to efficiently search for the pattern in the text.
* **Sudo Code**:

function computeLPSArray(pattern, lps):

length = 0

lps[0] = 0

i = 1

while i < pattern.length:

if pattern[i] == pattern[length]:

length += 1

lps[i] = length

i += 1

else:

if length != 0:

length = lps[length - 1]

else:

lps[i] = 0

i += 1

function KMPSearch(text, pattern):

n = text.length

m = pattern.length

lps = new array[m]

computeLPSArray(pattern, lps)

i = 0

j = 0

while i < n:

if pattern[j] == text[i]:

j += 1

i += 1

if j == m:

return true

else if i < n and pattern[j] != text[i]:

if j != 0:

j = lps[j - 1]

else:

i += 1

return false

* **Time Complexity**: O(n+m), where n is the length of the text and m is the length of the pattern.

### **Data Structures**

**AVL Tree**:

* **Purpose**: To maintain a balanced binary search tree for photos, allowing efficient insertion, deletion, and search operations.
* **Operations**:
  + Insertion: *O*(log*n*)
  + Search: *O*(log*n*)
  + Deletion: *O*(log*n*)
* **Space Complexity**: *O*(*n*)

**Trie**:

* **Purpose**: To efficiently search for photos by tags using prefix matching.
* **Operations**:
  + Insertion: *O*(*m*), where *m* is the length of the tag.
  + Search: *O*(*m*)
  + **Space Complexity**: *O*(*n*⋅*m*), where *n* is the number of tags and *m* is the average length of the tags.

**Priority Queue (Max Heap)**:

* **Purpose**: To efficiently retrieve the most recent or most popular photos.
* **Operations**:
  + Insertion: *O*(log*k*), where *k* is the size of the heap.
  + Extraction: *O*(log*k*)
* **Space Complexity**: *O*(*k*)

**Hash Map**:

* **Purpose**: To efficiently search for photos by location.
* **Operations**:
  + Insertion: *O*(1) on average.
  + Search: *O*(1) on average.
* **Space Complexity**: *O*(*n*), where *n* is the number of unique locations.

**Linked List**:

* **Purpose**: To maintain a sequential list of photos for operations like appending and removing.
* **Operations**:
  + Append: *O*(1)
  + Remove: *O*(*n*) in the worst case.
* **Space Complexity**: *O*(*n*)

#### **Time Complexity**

**Initialization**:

* Loading photos from the database involves iterating through all photos and inserting them into various data structures.

**Loading Photos**:

* *O*(*n*log*n*) for AVL Tree insertion, *O*(*n*⋅*m*) for Trie insertion, *O*(*n*log*k*) for Priority Queue insertion, *O*(*n*) for Hash Map insertion, and *O*(*n*) for Linked List insertion.
* **Overall**: *O*(*n*log*n*+*n*⋅*m*+*n*log*k*+*n*).

**Adding a Photo**:

* Inserting into AVL Tree: *O*(log*n*).
* Inserting into Trie: *O*(*m*).
* Inserting into Priority Queue: *O*(log*k*).
* Inserting into Hash Map: *O*(1).
* Inserting into Linked List: *O*(1).
* **Overall**: *O*(log*n*+*m*+log*k*).

**Viewing a Photo**:

* Incrementing view count and updating the database: *O*(1).
* Rebuilding the popularity tree and queue: *O*(*n*log*n*+*n*log*k*).
* **Overall**: *O*(*n*log*n*+*n*log*k*).

**Deleting a Photo**:

* Deleting from AVL Tree: *O*(log*n*).
* Deleting from Trie: *O*(*m*).
* Deleting from Priority Queue: *O*(log*k*).
* Deleting from Hash Map: *O*(1).
* Deleting from Linked List: *O*(*n*).
* Rebuilding data structures: *O*(*n*log*n*+*n*log*k*+*n*).
* **Overall**: *O*(*n*log*n*+*n*log*k*+*n*).

**Searching**:

* **By Location**: *O*(1) for Hash Map search, *O*(*n*) for iterating through results.
* **By Tag**: *O*(*n*⋅*m*) for Trie search.
* **By Date Range**: *O*(log*n*+*k*) for AVL Tree search.
* **By Prefix**: *O*(*m*) for Trie search.
* **By Description**: *O*(*n*⋅*m*) for KMP search.
* **Overall**: *O*(*n*⋅*m*) in the worst case.

**Sorting**:

* **QuickSort**: *O*(*n*log*n*).

**Retrieving Most Recent/Popular Photos**:

* Extracting from Priority Queue: *O*(*k*log*k*).

**Code**

// Photo Gallery System with SQLite and Custom Data Structures  
// Implements 5 manual data structures and 3 algorithms  
  
#include <iostream>  
#include <string>  
#include <ctime>  
#include <cstring>  
#include <sqlite3.h>  
#include <sstream>  
#include <cstdlib>  
#include <iomanip>  
#include <algorithm>  
#include <limits>  
  
using namespace std;  
  
// Forward declarations for data structures  
class AVLTree;  
class Trie;  
class PriorityQueue;  
class HashMap;  
class LinkedList;  
  
// Photo class to represent a photo with metadata  
class Photo {  
private:  
 int id;  
 string filename;  
 string location;  
 time\_t dateTime;  
 string description;  
 string tags[10];  
 int tagCount;  
 int viewCount;  
 int fileSize;  
  
public:  
 Photo(int id = -1, const string& filename = "", const string& location = "",   
 time\_t dateTime = time(nullptr), const string& description = "",   
 int fileSize = 0, int viewCount = 0)  
 : id(id), filename(filename), location(location), dateTime(dateTime),   
 description(description), fileSize(fileSize), viewCount(viewCount), tagCount(0) {}  
  
 // Getters  
 int getId() const { return id; }  
 string getFilename() const { return filename; }  
 string getLocation() const { return location; }  
 time\_t getDateTime() const { return dateTime; }  
 string getDescription() const { return description; }  
 string getTag(int index) const { return (index >= 0 && index < tagCount) ? tags[index] : ""; }  
 int getViewCount() const { return viewCount; }  
 int getFileSize() const { return fileSize; }  
 int getTagCount() const { return tagCount; }  
  
 // Setters  
 void setId(int id) { this->id = id; }  
 void setFilename(const string& filename) { this->filename = filename; }  
 void setLocation(const string& location) { this->location = location; }  
 void setDateTime(time\_t dateTime) { this->dateTime = dateTime; }  
 void setDescription(const string& description) { this->description = description; }  
 void setViewCount(int viewCount) { this->viewCount = viewCount; }  
 void setFileSize(int fileSize) { this->fileSize = fileSize; }  
   
 void incrementViewCount() { viewCount++; }  
   
 void addTag(const string& tag) {  
 if (tagCount < 10) {  
 // Check if tag already exists  
 for (int i = 0; i < tagCount; i++) {  
 if (tags[i] == tag) return;  
 }  
 tags[tagCount++] = tag;  
 }  
 }  
   
 bool hasTag(const string& tag) const {  
 for (int i = 0; i < tagCount; i++) {  
 if (tags[i] == tag) return true;  
 }  
 return false;  
 }  
   
 void setTags(const string& tagsStr) {  
 tagCount = 0;  
 string tag;  
 stringstream ss(tagsStr);  
 while (getline(ss, tag, ',') && tagCount < 10) {  
 // Trim whitespace  
 tag.erase(0, tag.find\_first\_not\_of(" \t\n\r\f\v"));  
 tag.erase(tag.find\_last\_not\_of(" \t\n\r\f\v") + 1);  
 if (!tag.empty()) {  
 tags[tagCount++] = tag;  
 }  
 }  
 }  
   
 string getTagsAsString() const {  
 string result;  
 for (int i = 0; i < tagCount; i++) {  
 if (i > 0) result += ", ";  
 result += tags[i];  
 }  
 return result;  
 }  
};  
  
// 1. AVL Tree implementation for balanced binary search tree  
class AVLNode {  
public:  
 Photo photo;  
 AVLNode\* left;  
 AVLNode\* right;  
 int height;  
   
 AVLNode(const Photo& p) : photo(p), left(nullptr), right(nullptr), height(1) {}  
};  
  
class AVLTree {  
private:  
 AVLNode\* root;  
   
 int height(AVLNode\* node) {  
 if (node == nullptr) return 0;  
 return node->height;  
 }  
   
 int getBalance(AVLNode\* node) {  
 if (node == nullptr) return 0;  
 return height(node->left) - height(node->right);  
 }  
   
 AVLNode\* rightRotate(AVLNode\* y) {  
 AVLNode\* x = y->left;  
 AVLNode\* T2 = x->right;  
   
 x->right = y;  
 y->left = T2;  
   
 y->height = max(height(y->left), height(y->right)) + 1;  
 x->height = max(height(x->left), height(x->right)) + 1;  
   
 return x;  
 }  
   
 AVLNode\* leftRotate(AVLNode\* x) {  
 AVLNode\* y = x->right;  
 AVLNode\* T2 = y->left;  
   
 y->left = x;  
 x->right = T2;  
   
 x->height = max(height(x->left), height(x->right)) + 1;  
 y->height = max(height(y->left), height(y->right)) + 1;  
   
 return y;  
 }  
   
 AVLNode\* insert(AVLNode\* node, const Photo& photo, bool byDate) {  
 // Standard BST insert  
 if (node == nullptr)  
 return new AVLNode(photo);  
   
 bool shouldGoLeft;  
 if (byDate) {  
 shouldGoLeft = difftime(photo.getDateTime(), node->photo.getDateTime()) < 0;  
 } else {  
 shouldGoLeft = photo.getViewCount() < node->photo.getViewCount();  
 }  
   
 if (shouldGoLeft)  
 node->left = insert(node->left, photo, byDate);  
 else  
 node->right = insert(node->right, photo, byDate);  
   
 // Update height  
 node->height = 1 + max(height(node->left), height(node->right));  
   
 // Get balance factor  
 int balance = getBalance(node);  
   
 // Left Left Case  
 if (balance > 1 && ((byDate && difftime(photo.getDateTime(), node->left->photo.getDateTime()) < 0) ||   
 (!byDate && photo.getViewCount() < node->left->photo.getViewCount()))) {  
 return rightRotate(node);  
 }  
   
 // Right Right Case  
 if (balance < -1 && ((byDate && difftime(photo.getDateTime(), node->right->photo.getDateTime()) >= 0) ||   
 (!byDate && photo.getViewCount() >= node->right->photo.getViewCount()))) {  
 return leftRotate(node);  
 }  
   
 // Left Right Case  
 if (balance > 1 && ((byDate && difftime(photo.getDateTime(), node->left->photo.getDateTime()) >= 0) ||   
 (!byDate && photo.getViewCount() >= node->left->photo.getViewCount()))) {  
 node->left = leftRotate(node->left);  
 return rightRotate(node);  
 }  
   
 // Right Left Case  
 if (balance < -1 && ((byDate && difftime(photo.getDateTime(), node->right->photo.getDateTime()) < 0) ||   
 (!byDate && photo.getViewCount() < node->right->photo.getViewCount()))) {  
 node->right = rightRotate(node->right);  
 return leftRotate(node);  
 }  
   
 return node;  
 }  
   
 void inOrderTraversal(AVLNode\* node, Photo\*& photos, int& index) {  
 if (node != nullptr) {  
 inOrderTraversal(node->left, photos, index);  
 photos[index++] = node->photo;  
 inOrderTraversal(node->right, photos, index);  
 }  
 }  
   
 void reverseInOrderTraversal(AVLNode\* node, Photo\*& photos, int& index) {  
 if (node != nullptr) {  
 reverseInOrderTraversal(node->right, photos, index);  
 photos[index++] = node->photo;  
 reverseInOrderTraversal(node->left, photos, index);  
 }  
 }  
   
 void clearTree(AVLNode\* node) {  
 if (node != nullptr) {  
 clearTree(node->left);  
 clearTree(node->right);  
 delete node;  
 }  
 }  
   
 // For date range search  
 void searchDateRange(AVLNode\* node, time\_t start, time\_t end, Photo\*\* results, int& count) {  
 if (node == nullptr) return;  
   
 // If current node is in range, check left subtree  
 if (difftime(node->photo.getDateTime(), start) >= 0) {  
 searchDateRange(node->left, start, end, results, count);  
 }  
   
 // Include current node if in range  
 if (difftime(node->photo.getDateTime(), start) >= 0 &&   
 difftime(end, node->photo.getDateTime()) >= 0) {  
 results[count++] = new Photo(node->photo);  
 }  
   
 // If current node is in range, check right subtree  
 if (difftime(end, node->photo.getDateTime()) >= 0) {  
 searchDateRange(node->right, start, end, results, count);  
 }  
 }  
   
public:  
 AVLTree() : root(nullptr) {}  
   
 ~AVLTree() {  
 clearTree(root);  
 }  
   
 void insert(const Photo& photo, bool byDate = true) {  
 root = insert(root, photo, byDate);  
 }  
   
 void getSortedPhotos(Photo\* photos, bool ascending = true) {  
 int index = 0;  
 if (ascending) {  
 inOrderTraversal(root, photos, index);  
 } else {  
 reverseInOrderTraversal(root, photos, index);  
 }  
 }  
   
 Photo\*\* searchByDateRange(time\_t start, time\_t end, int& count) {  
 Photo\*\* results = new Photo\*[100]; // Assuming max 100 photos  
 count = 0;  
 searchDateRange(root, start, end, results, count);  
 return results;  
 }  
   
 void rebuild(Photo\*\* photos, int count, bool byDate = true) {  
 clearTree(root);  
 root = nullptr;  
 for (int i = 0; i < count; i++) {  
 insert(\*photos[i], byDate);  
 }  
 }  
   
 int getSize(AVLNode\* node) {  
 if (node == nullptr) return 0;  
 return 1 + getSize(node->left) + getSize(node->right);  
 }  
   
 int getSize() {  
 return getSize(root);  
 }  
};  
  
// 2. Trie implementation for prefix searching  
class TrieNode {  
public:  
 TrieNode\* children[36]; // a-z and 0-9  
 bool isEndOfWord;  
 int photoIds[100]; // Store photo IDs that contain this tag/word  
 int photoCount;  
   
 TrieNode() : isEndOfWord(false), photoCount(0) {  
 for (int i = 0; i < 36; i++)  
 children[i] = nullptr;  
 }  
   
 ~TrieNode() {  
 for (int i = 0; i < 36; i++) {  
 if (children[i])  
 delete children[i];  
 }  
 }  
};  
  
class Trie {  
private:  
 TrieNode\*

root;  
   
 // Convert character to index (a-z, 0-9)  
 int charToIndex(char c) {  
 if (c >= 'a' && c <= 'z')  
 return c - 'a';  
 if (c >= 'A' && c <= 'Z')  
 return c - 'A';  
 if (c >= '0' && c <= '9')  
 return c - '0' + 26;  
 return -1; // Invalid character  
 }  
   
 void searchPrefix(TrieNode\* node, const string& prefix, int\* photoIds, int& count) {  
 if (node->isEndOfWord) {  
 for (int i = 0; i < node->photoCount; i++) {  
 bool exists = false;  
 for (int j = 0; j < count; j++) {  
 if (photoIds[j] == node->photoIds[i]) {  
 exists = true;  
 break;  
 }  
 }  
 if (!exists) {  
 photoIds[count++] = node->photoIds[i];  
 }  
 }  
 }  
   
 for (int i = 0; i < 36; i++) {  
 if (node->children[i]) {  
 searchPrefix(node->children[i], prefix, photoIds, count);  
 }  
 }  
 }  
   
public:  
 Trie() {  
 root = new TrieNode();  
 }  
   
 ~Trie() {  
 delete root;  
 }  
   
 void insert(const string& key, int photoId) {  
 TrieNode\* node = root;  
   
 for (size\_t i = 0; i < key.length(); i++) {  
 int index = charToIndex(key[i]);  
 if (index == -1) continue; // Skip invalid characters  
   
 if (!node->children[index])  
 node->children[index] = new TrieNode();  
   
 node = node->children[index];  
 }  
   
 // Mark last node as leaf and add photo ID  
 node->isEndOfWord = true;  
   
 // Add photo ID if not already present  
 bool exists = false;  
 for (int i = 0; i < node->photoCount; i++) {  
 if (node->photoIds[i] == photoId) {  
 exists = true;  
 break;  
 }  
 }  
   
 if (!exists && node->photoCount < 100) {  
 node->photoIds[node->photoCount++] = photoId;  
 }  
 }  
   
 int\* searchByPrefix(const string& prefix, int& count) {  
 TrieNode\* node = root;  
 int\* photoIds = new int[100]; // Assuming max 100 photos  
 count = 0;  
   
 // Navigate to the end of prefix  
 for (size\_t i = 0; i < prefix.length(); i++) {  
 int index = charToIndex(prefix[i]);  
 if (index == -1) continue; // Skip invalid characters  
   
 if (!node->children[index])  
 return photoIds; // Prefix not found  
   
 node = node->children[index];  
 }  
   
 // Find all words with the given prefix  
 searchPrefix(node, prefix, photoIds, count);  
 return photoIds;  
 }  
};  
  
// 3. Priority Queue (Max Heap) implementation for recent/popular photos  
class PriorityQueue {  
private:  
 Photo\* heap[100]; // Max heap  
 int size;  
 bool byViewCount; // Whether to prioritize by view count or date  
   
 void heapifyUp(int index) {  
 int parent = (index - 1) / 2;  
 while (index > 0) {  
 bool shouldSwap;  
 if (byViewCount) {  
 shouldSwap = heap[index]->getViewCount() > heap[parent]->getViewCount();  
 } else {  
 shouldSwap = difftime(heap[index]->getDateTime(), heap[parent]->getDateTime()) > 0;  
 }  
   
 if (shouldSwap) {  
 swap(heap[index], heap[parent]);  
 index = parent;  
 parent = (index - 1) / 2;  
 } else {  
 break;  
 }  
 }  
 }  
   
 void heapifyDown(int index) {  
 int maxIndex = index;  
 int left = 2 \* index + 1;  
 int right = 2 \* index + 2;  
   
 if (byViewCount) {  
 if (left < size && heap[left]->getViewCount() > heap[maxIndex]->getViewCount())  
 maxIndex = left;  
   
 if (right < size && heap[right]->getViewCount() > heap[maxIndex]->getViewCount())  
 maxIndex = right;  
 } else {  
 if (left < size && difftime(heap[left]->getDateTime(), heap[maxIndex]->getDateTime()) > 0)  
 maxIndex = left;  
   
 if (right < size && difftime(heap[right]->getDateTime(), heap[maxIndex]->getDateTime()) > 0)  
 maxIndex = right;  
 }  
   
 if (index != maxIndex) {  
 swap(heap[index], heap[maxIndex]);  
 heapifyDown(maxIndex);  
 }  
 }  
   
public:  
 PriorityQueue(bool byViewCount = false) : size(0), byViewCount(byViewCount) {}  
   
 void insert(Photo\* photo) {  
 if (size >= 100) return; // Heap is full  
   
 heap[size] = photo;  
 heapifyUp(size);  
 size++;  
 }  
   
 Photo\* extractMax() {  
 if (size == 0) return nullptr;  
   
 Photo\* result = heap[0];  
 heap[0] = heap[size - 1];  
 size--;  
 heapifyDown(0);  
   
 return result;  
 }  
   
 Photo\* peek() {  
 return (size > 0) ? heap[0] : nullptr;  
 }  
   
 bool isEmpty() {  
 return size == 0;  
 }  
   
 int getSize() {  
 return size;  
 }  
   
 void clear() {  
 size = 0;  
 }  
   
 Photo\*\* getAll(int& count) {  
 Photo\*\* result = new Photo\*[size];  
 count = size;  
   
 // Create a copy of the heap  
 PriorityQueue tempQueue(byViewCount);  
 for (int i = 0; i < size; i++) {  
 tempQueue.insert(heap[i]);  
 }  
   
 // Extract all elements  
 for (int i = 0; i < count; i++) {  
 result[i] = tempQueue.extractMax();  
 }  
   
 return result;  
 }  
};  
  
// 4. Hash Map implementation for location-based search  
class HashMapNode {  
public:  
 string key;  
 int photoIds[100];  
 int count;  
 HashMapNode\* next;  
   
 HashMapNode(const string& k, int photoId) : key(k), count(1), next(nullptr) {  
 photoIds[0] = photoId;  
 }  
};  
  
class HashMap {  
private:  
 static const int TABLE\_SIZE = 101;  
 HashMapNode\* table[TABLE\_SIZE];  
   
 int hashFunction(const string& key) {  
 int hash = 0;  
 for (char c : key) {  
 hash = (hash \* 31 + c) % TABLE\_SIZE;  
 }  
 return hash;  
 }  
   
public:  
 HashMap() {  
 for (int i = 0; i < TABLE\_SIZE; i++) {  
 table[i] = nullptr;  
 }  
 }  
   
 ~HashMap() {  
 for (int i = 0; i < TABLE\_SIZE; i++) {  
 HashMapNode\* current = table[i];  
 while (current != nullptr) {  
 HashMapNode\* next = current->next;  
 delete current;  
 current = next;  
 }  
 }  
 }  
   
 void insert(const string& key, int photoId) {  
 int index = hashFunction(key);  
   
 // Check if key already exists  
 HashMapNode\* current = table[index];  
 while (current != nullptr) {  
 if (current->key == key) {  
 // Check if photo ID already exists  
 for (int i = 0; i < current->count; i++) {  
 if (current->photoIds[i] == photoId) {  
 return; // Photo ID already exists  
 }  
 }  
   
 // Add photo ID  
 if (current->count < 100) {  
 current->photoIds[current->count++] = photoId;  
 }  
 return;  
 }  
 current = current->next;  
 }  
   
 // Create new node  
 HashMapNode\* newNode = new HashMapNode(key, photoId);  
 newNode->next = table[index];  
 table[index] = newNode;  
 }  
   
 int\* get(const string& key, int& count) {  
 int index = hashFunction(key);  
 HashMapNode\* current = table[index];  
   
 while (current != nullptr) {  
 if (current->key == key) {  
 count = current->count;  
 int\* result = new int[count];  
 for (int i = 0; i < count; i++) {  
 result[i] = current->photoIds[i];  
 }  
 return result;  
 }  
 current = current->next;  
 }  
   
 count = 0;  
 return new int[0];  
 }  
   
 void remove(const string& key) {  
 int index = hashFunction(key);  
 HashMapNode\* current = table[index];  
 HashMapNode\* prev = nullptr;  
   
 while (current != nullptr) {  
 if (current->key == key) {  
 if (prev == nullptr) {  
 table[index] = current->next;  
 } else {  
 prev->next = current->next;  
 }  
 delete current;  
 return;  
 }  
 prev = current;  
 current = current->next;  
 }  
 }  
   
 void getAllKeys(string\* keys, int& count) {  
 count = 0;  
 for (int i = 0; i < TABLE\_SIZE; i++) {  
 HashMapNode\* current = table[i];  
 while (current != nullptr) {  
 keys[count++] = current->key;  
 current = current->next;  
 }  
 }  
 }  
};  
  
// 5. Linked List implementation for sequential operations  
class ListNode {  
public:  
 Photo\* photo;  
 ListNode\* next;  
   
 ListNode(Photo\* p) : photo(p), next(nullptr) {}  
};  
  
class LinkedList {  
private:  
 ListNode\* head;  
 ListNode\* tail;  
 int size;  
   
public:  
 LinkedList() : head(nullptr), tail(nullptr), size(0) {}  
   
 ~LinkedList() {  
 ListNode\* current = head;  
 while (current != nullptr) {  
 ListNode\* next = current->next;  
 delete current; // Note: we don't delete the photo as it may be referenced elsewhere  
 current = next;  
 }  
 }  
   
 void append(Photo\* photo) {  
 ListNode\*

newNode = new ListNode(photo);  
 if (head == nullptr) {  
 head = newNode;  
 tail = newNode;  
 } else {  
 tail->next = newNode;  
 tail = newNode;  
 }  
 size++;  
 }  
   
 void insertAtBeginning(Photo\* photo) {  
 ListNode\* newNode = new ListNode(photo);  
 if (head == nullptr) {  
 head = newNode;  
 tail = newNode;  
 } else {  
 newNode->next = head;  
 head = newNode;  
 }  
 size++;  
 }  
   
 Photo\* getAt(int index) {  
 if (index < 0 || index >= size) return nullptr;  
   
 ListNode\* current = head;  
 for (int i = 0; i < index; i++) {  
 current = current->next;  
 }  
 return current->photo;  
 }  
   
 void removeAt(int index) {  
 if (index < 0 || index >= size) return;  
   
 if (index == 0) {  
 ListNode\* temp = head;  
 head = head->next;  
 if (head == nullptr) tail = nullptr;  
 delete temp;  
 } else {  
 ListNode\* current = head;  
 for (int i = 0; i < index - 1; i++) {  
 current = current->next;  
 }  
 ListNode\* temp = current->next;  
 current->next = temp->next;  
 if (temp == tail) tail = current;  
 delete temp;  
 }  
 size--;  
 }  
   
 int getSize() const {  
 return size;  
 }  
   
 void getAllPhotos(Photo\*\* photos) {  
 ListNode\* current = head;  
 int i = 0;  
 while (current != nullptr) {  
 photos[i++] = current->photo;  
 current = current->next;  
 }  
 }  
};  
  
  
  
  
  
  
  
  
  
  
// Helper functions for time conversions  
time\_t stringToTime(const string& dateString) {  
 struct tm tm = {};  
 tm.tm\_year = stoi(dateString.substr(0,4)) - 1900;  
 tm.tm\_mon = stoi(dateString.substr(5,2)) - 1;  
 tm.tm\_mday = stoi(dateString.substr(8,2));  
 return mktime(&tm);  
}  
  
string timeToString(time\_t time) {  
 char buffer[80];  
 struct tm\* timeinfo = localtime(&time);  
 strftime(buffer, sizeof(buffer), "%Y-%m-%d", timeinfo);  
 return string(buffer);  
}  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
// Algorithm 1: Quick Sort implementation for sorting photos  
// Custom QuickSort for Photo arrays (sorting by view count, size, or date)  
enum SortType { BY\_DATE, BY\_SIZE, BY\_VIEWS };  
  
int partitionArray(Photo\*\* photos, int low, int high, SortType sortType) {  
 Photo\* pivot = photos[high];  
 int i = low - 1;  
   
 for (int j = low; j <= high - 1; j++) {  
 bool shouldSwap = false;  
   
 switch (sortType) {  
 case BY\_DATE:  
 shouldSwap = difftime(photos[j]->getDateTime(), pivot->getDateTime()) > 0;  
 break;  
 case BY\_SIZE:  
 shouldSwap = photos[j]->getFileSize() > pivot->getFileSize();  
 break;  
 case BY\_VIEWS:  
 shouldSwap = photos[j]->getViewCount() > pivot->getViewCount();  
 break;  
 }  
   
 if (shouldSwap) {  
 i++;  
 swap(photos[i], photos[j]);  
 }  
 }  
   
 swap(photos[i + 1], photos[high]);  
 return i + 1;  
}  
  
void quickSort(Photo\*\* photos, int low, int high, SortType sortType) {  
 if (low < high) {  
 int pi = partitionArray(photos, low, high, sortType);  
   
 quickSort(photos, low, pi - 1, sortType);  
 quickSort(photos, pi + 1, high, sortType);  
 }  
}  
  
  
  
  
  
  
  
  
  
  
  
  
  
// Algorithm 2: Binary Search for photos within a date range  
int binarySearchDate(Photo\*\* photos, int left, int right, time\_t date) {  
 if (right >= left) {  
 int mid = left + (right - left) / 2;  
   
 if (difftime(photos[mid]->getDateTime(), date) == 0)  
 return mid;  
   
 if (difftime(photos[mid]->getDateTime(), date) > 0)  
 return binarySearchDate(photos, left, mid - 1, date);  
   
 return binarySearchDate(photos, mid + 1, right, date);  
 }  
   
 return -1;  
}  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
// Algorithm 3: KMP String Matching Algorithm for searching text in descriptions  
void computeLPSArray(const string& pattern, int\* lps) {  
 int length = 0;  
 lps[0] = 0;  
 int i = 1;  
 int m = pattern.length();  
   
 while (i < m) {  
 if (pattern[i] == pattern[length]) {  
 length++;  
 lps[i] = length;  
 i++;  
 } else {  
 if (length != 0) {  
 length = lps[length - 1];  
 } else {  
 lps[i] = 0;  
 i++;  
 }  
 }  
 }  
}  
  
bool KMPSearch(const string& text, const string& pattern) {  
 int n = text.length();  
 int m = pattern.length();  
   
 if (m == 0) return true;  
 if (n == 0) return false;  
   
 int\* lps = new int[m];  
 computeLPSArray(pattern, lps);  
   
 int i = 0; // index for text  
 int j = 0; // index for pattern  
   
 while (i < n) {  
 if (pattern[j] == text[i]) {  
 j++;  
 i++;  
 }  
   
 if (j == m) {  
 delete[] lps;  
 return true;  
 } else if (i < n && pattern[j] != text[i]) {  
 if (j != 0) {  
 j = lps[j - 1];  
 } else {  
 i++;  
 }  
 }  
 }  
   
 delete[] lps;  
 return false;  
}  
  
// Database callback function  
static int callback(void\* data, int argc, char\*\* argv, char\*\* azColName) {  
 int\* id = static\_cast<int\*>(data);  
 \*id = atoi(argv[0]);  
 return 0;  
}  
  
// Photo Gallery System class  
class PhotoGallerySystem {  
private:  
 sqlite3\* db;  
 Photo\* photos[1000];  
 int photoCount;  
   
 AVLTree dateTree;  
 AVLTree popularityTree;  
 Trie tagTrie;  
 PriorityQueue recentQueue;  
 PriorityQueue popularQueue;  
 HashMap locationMap;  
 LinkedList photoList;  
   
 // Initialize database  
 bool initDatabase() {  
 int rc = sqlite3\_open("photo\_gallery.db", &db);  
 if (rc) {  
 cerr << "Can't open database: " << sqlite3\_errmsg(db) << endl;  
 return false;  
 }  
   
 // Create tables if they don't exist  
 const char\* createPhotoTable =   
 "CREATE TABLE IF NOT EXISTS photos("  
 "id INTEGER PRIMARY KEY AUTOINCREMENT,"  
 "filename TEXT NOT NULL,"  
 "location TEXT,"  
 "date\_time INTEGER,"  
 "description TEXT,"  
 "file\_size INTEGER,"  
 "view\_count INTEGER DEFAULT 0);";  
   
 const char\* createTagTable =   
 "CREATE TABLE IF NOT EXISTS tags("  
 "id INTEGER PRIMARY KEY AUTOINCREMENT,"  
 "photo\_id INTEGER,"  
 "tag TEXT NOT NULL,"  
 "FOREIGN KEY(photo\_id) REFERENCES photos(id));";  
   
 char\* errMsg;  
 rc = sqlite3\_exec(db, createPhotoTable, nullptr, nullptr, &errMsg);  
 if (rc != SQLITE\_OK) {  
 cerr << "SQL error: " << errMsg << endl;  
 sqlite3\_free(errMsg);  
 return false;  
 }  
   
 rc = sqlite3\_exec(db, createTagTable, nullptr, nullptr, &errMsg);  
 if (rc != SQLITE\_OK) {  
 cerr << "SQL error: " << errMsg << endl;  
 sqlite3\_free(errMsg);  
 return false;  
 }  
   
 return true;  
 }  
   
 // Load all photos from database  
 void loadPhotosFromDB() {  
 photoCount = 0;  
   
 // Clear existing data structures  
 recentQueue.clear();  
 popularQueue.clear();  
   
 // SQL to retrieve all photos  
 const char\* sql = "SELECT p.id, p.filename, p.location, p.date\_time, p.description, "  
 "p.file\_size, p.view\_count, GROUP\_CONCAT(t.tag, ',') as tags "  
 "FROM photos p LEFT JOIN tags t ON p.id = t.photo\_id "  
 "GROUP BY p.id;";  
   
 sqlite3\_stmt\* stmt;  
 int rc = sqlite3\_prepare\_v2(db, sql, -1, &stmt, nullptr);  
   
 if (rc != SQLITE\_OK) {  
 cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << endl;  
 return;  
 }  
   
 while (sqlite3\_step(stmt) == SQLITE\_ROW) {  
 int id = sqlite3\_column\_int(stmt, 0);  
 string filename = (char\*)sqlite3\_column\_text(stmt, 1);  
   
 string location = "";  
 if (sqlite3\_column\_text(stmt, 2) != nullptr) {  
 location = (char\*)sqlite3\_column\_text(stmt, 2);  
 }  
   
 time\_t dateTime = sqlite3\_column\_int64(stmt, 3);  
   
 string description = "";  
 if (sqlite3\_column\_text(stmt, 4) != nullptr) {  
 description = (char\*)sqlite3\_column\_text(stmt, 4);  
 }  
   
 int fileSize = sqlite3\_column\_int(stmt, 5);  
 int viewCount = sqlite3\_column\_int(stmt, 6);  
   
 Photo\* photo = new Photo(id, filename, location, dateTime, description, fileSize, viewCount);  
   
 // Add tags if available  
 if (sqlite3\_column\_text(stmt, 7) != nullptr) {  
 string tagsStr = (char\*)sqlite3\_column\_text(stmt, 7);  
 photo->setTags(tagsStr);  
   
 // Add tags to trie  
 stringstream ss(tagsStr);  
 string tag;  
 while (getline(ss, tag, ',')) {  
 // Trim whitespace  
 tag.erase(0, tag.find\_first\_not\_of(" \t\n\r\f\v"));  
 tag.erase(tag.find\_last\_not\_of(" \t\n\r\f\v") + 1);  
 if (!tag.empty()) {  
 tagTrie.insert(tag, id);  
 }  
 }  
 }  
   
 // Add to arrays and data structures  
 photos[photoCount++] = photo;  
 photoList.append(photo);  
 dateTree.insert(\*photo);

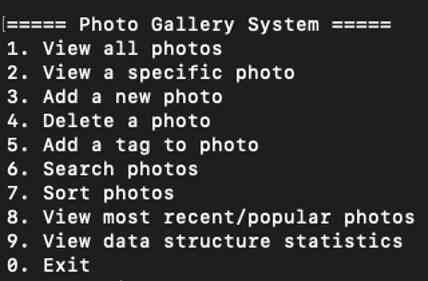
popularityTree.insert(\*photo, false);  
 recentQueue.insert(photo);  
 popularQueue.insert(photo);  
 locationMap.insert(location, id);  
 }  
   
 sqlite3\_finalize(stmt);  
 }  
   
  
  
  
  
  
  
 // Save photo to database  
 int savePhotoToDB(Photo& photo) {  
 const char\* sql = "INSERT INTO photos (filename, location, date\_time, description, file\_size, view\_count) "  
 "VALUES (?, ?, ?, ?, ?, ?);";  
   
 sqlite3\_stmt\* stmt;  
 int rc = sqlite3\_prepare\_v2(db, sql, -1, &stmt, nullptr);  
   
 if (rc != SQLITE\_OK) {  
 cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << endl;  
 return -1;  
 }  
   
 sqlite3\_bind\_text(stmt, 1, photo.getFilename().c\_str(), -1, SQLITE\_STATIC);  
 sqlite3\_bind\_text(stmt, 2, photo.getLocation().c\_str(), -1, SQLITE\_STATIC);  
 sqlite3\_bind\_int64(stmt, 3, photo.getDateTime());  
 sqlite3\_bind\_text(stmt, 4, photo.getDescription().c\_str(), -1, SQLITE\_STATIC);  
 sqlite3\_bind\_int(stmt, 5, photo.getFileSize());  
 sqlite3\_bind\_int(stmt, 6, photo.getViewCount());  
   
 rc = sqlite3\_step(stmt);  
 if (rc != SQLITE\_DONE) {  
 cerr << "Execution failed: " << sqlite3\_errmsg(db) << endl;  
 sqlite3\_finalize(stmt);  
 return -1;  
 }  
   
 int photoId = sqlite3\_last\_insert\_rowid(db);  
 photo.setId(photoId);  
   
 sqlite3\_finalize(stmt);  
   
 // Save tags  
 for (int i = 0; i < photo.getTagCount(); i++) {  
 string tag = photo.getTag(i);  
   
 const char\* tagSql = "INSERT INTO tags (photo\_id, tag) VALUES (?, ?);";  
   
 rc = sqlite3\_prepare\_v2(db, tagSql, -1, &stmt, nullptr);  
 if (rc != SQLITE\_OK) {  
 cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << endl;  
 continue;  
 }  
   
 sqlite3\_bind\_int(stmt, 1, photoId);  
 sqlite3\_bind\_text(stmt, 2, tag.c\_str(), -1, SQLITE\_STATIC);  
   
 rc = sqlite3\_step(stmt);  
 if (rc != SQLITE\_DONE) {  
 cerr << "Execution failed: " << sqlite3\_errmsg(db) << endl;  
 }  
   
 sqlite3\_finalize(stmt);  
 }  
   
 return photoId;  
 }  
   
 // Update photo in database  
 bool updatePhotoInDB(const Photo& photo) {  
 const char\* sql = "UPDATE photos SET filename = ?, location = ?, date\_time = ?, "  
 "description = ?, file\_size = ?, view\_count = ? WHERE id = ?;";  
   
 sqlite3\_stmt\* stmt;  
 int rc = sqlite3\_prepare\_v2(db, sql, -1, &stmt, nullptr);  
   
 if (rc != SQLITE\_OK) {  
 cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << endl;  
 return false;  
 }  
   
 sqlite3\_bind\_text(stmt, 1, photo.getFilename().c\_str(), -1, SQLITE\_STATIC);  
 sqlite3\_bind\_text(stmt, 2, photo.getLocation().c\_str(), -1, SQLITE\_STATIC);  
 sqlite3\_bind\_int64(stmt, 3, photo.getDateTime());  
 sqlite3\_bind\_text(stmt, 4, photo.getDescription().c\_str(), -1, SQLITE\_STATIC);  
 sqlite3\_bind\_int(stmt, 5, photo.getFileSize());  
 sqlite3\_bind\_int(stmt, 6, photo.getViewCount());  
 sqlite3\_bind\_int(stmt, 7, photo.getId());  
   
 rc = sqlite3\_step(stmt);  
 if (rc != SQLITE\_DONE) {  
 cerr << "Execution failed: " << sqlite3\_errmsg(db) << endl;  
 sqlite3\_finalize(stmt);  
 return false;  
 }  
   
 sqlite3\_finalize(stmt);  
   
 // Delete existing tags and insert new ones  
 const char\* deleteSql = "DELETE FROM tags WHERE photo\_id = ?;";  
 rc = sqlite3\_prepare\_v2(db, deleteSql, -1, &stmt, nullptr);  
   
 if (rc != SQLITE\_OK) {  
 cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << endl;  
 return false;  
 }  
   
 sqlite3\_bind\_int(stmt, 1, photo.getId());  
   
 rc = sqlite3\_step(stmt);  
 if (rc != SQLITE\_DONE) {  
 cerr << "Execution failed: " << sqlite3\_errmsg(db) << endl;  
 sqlite3\_finalize(stmt);  
 return false;  
 }  
   
 sqlite3\_finalize(stmt);  
   
 // Insert new tags  
 for (int i = 0; i < photo.getTagCount(); i++) {  
 string tag = photo.getTag(i);  
   
 const char\* tagSql = "INSERT INTO tags (photo\_id, tag) VALUES (?, ?);";  
   
 rc = sqlite3\_prepare\_v2(db, tagSql, -1, &stmt, nullptr);  
 if (rc != SQLITE\_OK) {  
 cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << endl;  
 continue;  
 }  
   
 sqlite3\_bind\_int(stmt, 1, photo.getId());  
 sqlite3\_bind\_text(stmt, 2, tag.c\_str(), -1, SQLITE\_STATIC);  
   
 rc = sqlite3\_step(stmt);  
 if (rc != SQLITE\_DONE) {  
 cerr << "Execution failed: " << sqlite3\_errmsg(db) << endl;  
 }  
   
 sqlite3\_finalize(stmt);  
 }  
   
 return true;  
 }  
   
 // Delete photo from database  
 bool deletePhotoFromDB(int photoId) {  
 // First delete tags  
 const char\* deleteTags = "DELETE FROM tags WHERE photo\_id = ?;";  
 sqlite3\_stmt\* stmt;  
 int rc = sqlite3\_prepare\_v2(db, deleteTags, -1, &stmt, nullptr);  
   
 if (rc != SQLITE\_OK) {  
 cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << endl;  
 return false;  
 }  
   
 sqlite3\_bind\_int(stmt, 1, photoId);  
   
 rc = sqlite3\_step(stmt);  
 if (rc != SQLITE\_DONE) {  
 cerr << "Execution failed: " << sqlite3\_errmsg(db) << endl;  
 sqlite3\_finalize(stmt);  
 return false;  
 }  
   
 sqlite3\_finalize(stmt);  
   
 // Then delete photo  
 const char\* deletePhoto = "DELETE FROM photos WHERE id = ?;";  
 rc = sqlite3\_prepare\_v2(db, deletePhoto, -1, &stmt, nullptr);  
   
 if (rc != SQLITE\_OK) {  
 cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << endl;  
 return false;  
 }  
   
 sqlite3\_bind\_int(stmt, 1, photoId);  
   
 rc = sqlite3\_step(stmt);  
 if (rc != SQLITE\_DONE) {  
 cerr << "Execution failed: " << sqlite3\_errmsg(db) << endl;  
 sqlite3\_finalize(stmt);  
 return false;  
 }  
   
 sqlite3\_finalize(stmt);  
 return true;  
 }  
  
public:  
 PhotoGallerySystem() : photoCount(0) {  
 // Initialize database  
 if (!initDatabase()) {  
 cerr << "Failed to initialize database" << endl;  
 exit(1);  
 }  
   
 // Load photos from database  
 loadPhotosFromDB();  
 }  
   
 ~PhotoGallerySystem() {  
 // Free memory for photos  
 for (int i = 0; i < photoCount; i++) {  
 delete photos[i];  
 }  
   
 // Close database  
 sqlite3\_close(db);  
 }  
   
 // Add a new photo  
 bool addPhoto(const string& filename, const string& location, const string& dateStr,   
 const string& description, const string& tagsStr, int fileSize) {  
 time\_t dateTime = stringToTime(dateStr);  
   
 Photo photo(-1, filename, location, dateTime, description, fileSize, 0);  
 photo.setTags(tagsStr);  
   
 // Save to database  
 int photoId = savePhotoToDB(photo);  
 if (photoId == -1) {  
 return false;  
 }  
   
 // Create a new photo with the ID  
 Photo\* newPhoto = new Photo(photoId, filename, location, dateTime, description, fileSize, 0);  
 newPhoto->setTags(tagsStr);  
   
 // Add to data structures  
 photos[photoCount++] = newPhoto;  
 photoList.append(newPhoto);  
 dateTree.insert(\*newPhoto);  
 popularityTree.insert(\*newPhoto, false);  
 recentQueue.insert(newPhoto);  
 popularQueue.insert(newPhoto);  
 locationMap.insert(location, photoId);  
   
 // Add tags to trie  
 for (int i = 0; i < newPhoto->getTagCount(); i++) {  
 tagTrie.insert(newPhoto->getTag(i), photoId);  
 }  
   
 return true;  
 }  
   
 // View a photo (increment view count)  
 bool viewPhoto(int index) {  
 if (index < 0 || index >= photoCount) {  
 return false;  
 }  
   
 photos[index]->incrementViewCount();  
   
 // Update in database  
 updatePhotoInDB(\*photos[index]);  
   
 // Rebuild popularity tree & queue  
 popularityTree.rebuild(photos, photoCount, false);  
   
 // Clear and rebuild popular queue  
 popularQueue.clear();  
 for (int i = 0; i < photoCount; i++) {  
 popularQueue.insert(photos[i]);  
 }  
   
 return true;  
 }  
   
 // Delete a photo  
 bool deletePhoto(int index) {  
 if (index < 0 || index >= photoCount) {  
 return false;  
 }  
   
 int photoId = photos[index]->getId();  
   
 // Delete from database  
 if (!deletePhotoFromDB(photoId)) {  
 return false;  
 }  
   
 // Remove the photo from memory  
 delete photos[index];  
   
 // Shift all photos  
 for (int i = index; i < photoCount - 1; i++) {  
 photos[i] = photos[i + 1];  
 }  
 photoCount--;  
   
 // Rebuild all data structures (simple approach)  
 dateTree.rebuild(photos, photoCount);  
 popularityTree.rebuild(photos, photoCount, false);  
   
 // Rebuild queues  
 recentQueue.clear();  
 popularQueue.clear();

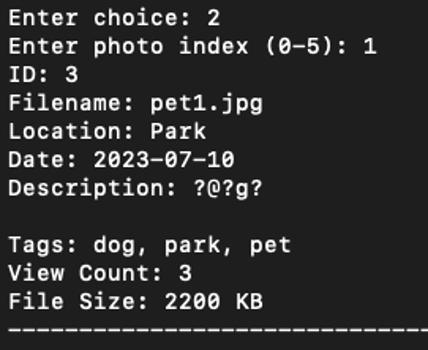
for (int i = 0; i < photoCount; i++) {  
 recentQueue.insert(photos[i]);  
 popularQueue.insert(photos[i]);  
 }  
   
 // For simplicity, rebuild the LinkedList  
 // (In a real implementation, we'd find and remove the specific node)  
 photoList = LinkedList();  
 for (int i = 0; i < photoCount; i++) {  
 photoList.append(photos[i]);  
 }  
   
 return true;  
 }  
   
 // Search by location  
 void searchByLocation(const string& location, Photo\*\* results, int& count) {  
 count = 0;  
 int\* photoIds;  
 int idCount;  
   
 photoIds = locationMap.get(location, idCount);  
   
 for (int i = 0; i < idCount; i++) {  
 for (int j = 0; j < photoCount; j++) {  
 if (photos[j]->getId() == photoIds[i]) {  
 results[count++] = photos[j];  
 break;  
 }  
 }  
 }  
   
 delete[] photoIds;  
 }  
   
 // Search by tag  
 void searchByTag(const string& tag, Photo\*\* results, int& count) {  
 count = 0;  
   
 for (int i = 0; i < photoCount; i++) {  
 if (photos[i]->hasTag(tag)) {  
 results[count++] = photos[i];  
 }  
 }  
 }  
   
 // Search by date range  
 void searchByDateRange(const string& startDateStr, const string& endDateStr, Photo\*\* results, int& count) {  
 time\_t startDate = stringToTime(startDateStr);  
 time\_t endDate = stringToTime(endDateStr);  
   
 count = 0;  
 Photo\*\* dateResults = dateTree.searchByDateRange(startDate, endDate, count);  
   
 for (int i = 0; i < count; i++) {  
 results[i] = dateResults[i];  
 }  
   
 delete[] dateResults;  
 }  
   
 // Search by keyword prefix using Trie  
 void searchByPrefix(const string& prefix, Photo\*\* results, int& count) {  
 int\* photoIds = tagTrie.searchByPrefix(prefix, count);  
   
 int resultCount = 0;  
 for (int i = 0; i < count; i++) {  
 for (int j = 0; j < photoCount; j++) {  
 if (photos[j]->getId() == photoIds[i]) {  
 bool exists = false;  
 for (int k = 0; k < resultCount; k++) {  
 if (results[k]->getId() == photos[j]->getId()) {  
 exists = true;  
 break;  
 }  
 }  
 if (!exists) {  
 results[resultCount++] = photos[j];  
 }  
 break;  
 }  
 }  
 }  
   
 count = resultCount;  
 delete[] photoIds;  
 }  
   
 // Search by description text using KMP algorithm  
 void searchByDescription(const string& text, Photo\*\* results, int& count) {  
 count = 0;  
   
 for (int i = 0; i < photoCount; i++) {  
 string description = photos[i]->getDescription();  
 // Convert both to lowercase for case-insensitive search  
 string lowerDesc = description;  
 string lowerText = text;  
   
 for (size\_t j = 0; j < lowerDesc.length(); j++) {  
 lowerDesc[j] = tolower(lowerDesc[j]);  
 }  
   
 for (size\_t j = 0; j < lowerText.length(); j++) {  
 lowerText[j] = tolower(lowerText[j]);  
 }  
   
 if (KMPSearch(lowerDesc, lowerText)) {  
 results[count++] = photos[i];  
 }  
 }  
 }  
   
 // Sort photos by date  
 void sortByDate(Photo\*\* results, bool descending = true) {  
 for (int i = 0; i < photoCount; i++) {  
 results[i] = photos[i];  
 }  
   
 quickSort(results, 0, photoCount - 1, BY\_DATE);  
   
 if (!descending) {  
 // Reverse the array  
 for (int i = 0; i < photoCount / 2; i++) {  
 swap(results[i], results[photoCount - i - 1]);  
 }  
 }  
 }  
   
 // Sort photos by size  
 void sortBySize(Photo\*\* results, bool descending = true) {  
 for (int i = 0; i < photoCount; i++) {  
 results[i] = photos[i];  
 }  
   
 quickSort(results, 0, photoCount - 1, BY\_SIZE);  
   
 if (!descending) {  
 // Reverse the array  
 for (int i = 0; i < photoCount / 2; i++) {  
 swap(results[i], results[photoCount - i - 1]);  
 }  
 }  
 }  
   
 // Sort photos by popularity (view count)  
 void sortByPopularity(Photo\*\* results, bool descending = true) {  
 for (int i = 0; i < photoCount; i++) {  
 results[i] = photos[i];  
 }  
   
 quickSort(results, 0, photoCount - 1, BY\_VIEWS);  
   
 if (!descending) {  
 // Reverse the array  
 for (int i = 0; i < photoCount / 2; i++) {  
 swap(results[i], results[photoCount - i - 1]);  
 }  
 }  
 }  
   
 // Get most recent photos using priority queue  
 void getMostRecentPhotos(Photo\*\* results, int& count, int limit = 5) {  
 PriorityQueue tempQueue = recentQueue;  
 count = min(limit, tempQueue.getSize());  
   
 for (int i = 0; i < count; i++) {  
 results[i] = tempQueue.extractMax();  
 }  
 }  
   
 // Get most popular photos using priority queue  
 void getMostPopularPhotos(Photo\*\* results, int& count, int limit = 5) {  
 PriorityQueue tempQueue = popularQueue;  
 count = min(limit, tempQueue.getSize());  
   
 for (int i = 0; i < count; i++) {  
 results[i] = tempQueue.extractMax();  
 }  
 }  
   
 // Display photo details  
 void displayPhoto(const Photo\* photo) {  
 cout << "ID: " << photo->getId() << endl;  
 cout << "Filename: " << photo->getFilename() << endl;  
 cout << "Location: " << photo->getLocation() << endl;  
   
 time\_t dateTime = photo->getDateTime();  
 struct tm\* timeinfo = localtime(&dateTime);  
 char buffer[80];  
 strftime(buffer, sizeof(buffer), "%Y-%m-%d", timeinfo);  
   
 cout << "Date: " << buffer << endl;  
 cout << "Description: " << photo->getDescription() << endl;  
   
 cout << "Tags: ";  
 for (int i = 0; i < photo->getTagCount(); i++) {  
 if (i > 0) cout << ", ";  
 cout << photo->getTag(i);  
 }  
 cout << endl;  
   
 cout << "View Count: " << photo->getViewCount() << endl;  
 cout << "File Size: " << photo->getFileSize() << " KB" << endl;  
 cout << "------------------------------" << endl;  
 }  
   
 // Display all photos  
 void displayAllPhotos() {  
 cout << "\n===== All Photos (" << photoCount << ") =====" << endl;  
 for (int i = 0; i < photoCount; i++) {  
 cout << "[" << i << "] ";  
 displayPhoto(photos[i]);  
 }  
 }  
   
 // Get photo count  
 int getPhotoCount() const {  
 return photoCount;  
 }  
   
 // Get photo by index  
 Photo\* getPhoto(int index) {  
 if (index >= 0 && index < photoCount) {  
 return photos[index];  
 }  
 return nullptr;  
 }  
   
 // Get all photos  
 void getAllPhotos(Photo\*\* results) {  
 for (int i = 0; i < photoCount; i++) {  
 results[i] = photos[i];  
 }  
 }  
   
 // Add tag to photo  
 bool addTagToPhoto(int index, const string& tag) {  
 if (index < 0 || index >= photoCount) {  
 return false;  
 }  
   
 photos[index]->addTag(tag);  
 updatePhotoInDB(\*photos[index]);  
   
 // Add to trie  
 tagTrie.insert(tag, photos[index]->getId());  
   
 return true;  
 }  
   
 // Get unique locations  
 void getUniqueLocations(string\* locations, int& count) {  
 locationMap.getAllKeys(locations, count);  
 }  
   
 // Get data structure stats  
 void getDataStructureStats() {  
 cout << "\n===== Data Structure Statistics =====" << endl;  
 cout << "Total Photos: " << photoCount << endl;  
 cout << "Date Tree Size: " << dateTree.getSize() << endl;  
 cout << "Recent Queue Size: " << recentQueue.getSize() << endl;  
 cout << "Popular Queue Size: " << popularQueue.getSize() << endl;  
 cout << "Photo List Size: " << photoList.getSize() << endl;  
   
 // Count unique locations  
 string locations[100];  
 int locationCount = 0;  
 locationMap.getAllKeys(locations, locationCount);  
 cout << "Unique Locations: " << locationCount << endl;  
 }  
};  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
int main() {  
 // Initialize the Photo Gallery System  
 PhotoGallerySystem gallery;  
   
 // Add some sample photos if the database is empty  
 if (gallery.getPhotoCount() == 0) {  
 cout << "Initializing database with sample photos..." << endl;  
   
 gallery.addPhoto("vacation1.jpg", "Paris", "2023-06-15",   
 "Eiffel Tower at sunset", "vacation,paris,landmark", 2500);  
   
 gallery.addPhoto("family1.jpg", "Home", "2023-05-20",   
 "Family dinner celebration", "family,dinner,home", 1800);  
   
 gallery.addPhoto("pet1.jpg", "Park", "2023-07-10",   
 "My dog playing in the park", "pet,dog,park", 2200);  
   
 gallery.addPhoto("vacation2.jpg", "Rome", "2023-06-18",   
 "Colosseum tour", "vacation,rome,landmark", 3000);  
   
 gallery.addPhoto("work1.jpg", "Office", "2023-04-25",   
 "Team building event", "work,team,office", 1500);  
   
 cout << "Sample photos added successfully!" << endl;  
 }  
   
 // Main menu  
 int choice;  
 bool running = true;  
   
 while (running) {  
 cout << "\n===== Photo Gallery System ====="

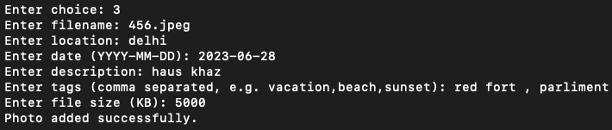
<< endl;  
 cout << "1. View all photos" << endl;  
 cout << "2. View a specific photo" << endl;  
 cout << "3. Add a new photo" << endl;  
 cout << "4. Delete a photo" << endl;  
 cout << "5. Add a tag to photo" << endl;  
 cout << "6. Search photos" << endl;  
 cout << "7. Sort photos" << endl;  
 cout << "8. View most recent/popular photos" << endl;  
 cout << "9. View data structure statistics" << endl;  
 cout << "0. Exit" << endl;  
 cout << "Enter choice: ";  
   
 if (!(cin >> choice)) {  
 cin.clear();  
 cin.ignore(numeric\_limits<streamsize>::max(), '\n');  
 cout << "Invalid input. Please enter a number." << endl;  
 continue;  
 }  
 cin.ignore();  
   
 switch (choice) {  
 case 0:  
 running = false;  
 break;  
   
 case 1:  
 gallery.displayAllPhotos();  
 break;  
   
 case 2: {  
 if (gallery.getPhotoCount() == 0) {  
 cout << "No photos available." << endl;  
 break;  
 }  
   
 int index;  
 cout << "Enter photo index (0-" << gallery.getPhotoCount() - 1 << "): ";  
 cin >> index;  
 cin.ignore();  
   
 if (index >= 0 && index < gallery.getPhotoCount()) {  
 gallery.viewPhoto(index);  
 gallery.displayPhoto(gallery.getPhoto(index));  
 } else {  
 cout << "Invalid index." << endl;  
 }  
 break;  
 }  
   
 case 3: {  
 string filename, location, dateStr, description, tagsStr;  
 int fileSize;  
   
 cout << "Enter filename: ";  
 getline(cin, filename);  
   
 cout << "Enter location: ";  
 getline(cin, location);  
   
 cout << "Enter date (YYYY-MM-DD): ";  
 getline(cin, dateStr);  
   
 cout << "Enter description: ";  
 getline(cin, description);  
   
 cout << "Enter tags (comma separated, e.g. vacation,beach,sunset): ";  
 getline(cin, tagsStr);  
   
 cout << "Enter file size (KB): ";  
 cin >> fileSize;  
 cin.ignore();  
   
 if (gallery.addPhoto(filename, location, dateStr, description, tagsStr, fileSize)) {  
 cout << "Photo added successfully." << endl;  
 } else {  
 cout << "Failed to add photo." << endl;  
 }  
 break;  
 }  
   
 case 4: {  
 if (gallery.getPhotoCount() == 0) {  
 cout << "No photos available to delete." << endl;  
 break;  
 }  
   
 int index;  
 cout << "Enter index of photo to delete (0-" << gallery.getPhotoCount() - 1 << "): ";  
 cin >> index;  
 cin.ignore();  
   
 if (gallery.deletePhoto(index)) {  
 cout << "Photo deleted successfully." << endl;  
 } else {  
 cout << "Failed to delete photo." << endl;  
 }  
 break;  
 }  
   
 case 5: {  
 if (gallery.getPhotoCount() == 0) {  
 cout << "No photos available." << endl;  
 break;  
 }  
   
 int index;  
 string tag;  
   
 cout << "Enter photo index (0-" << gallery.getPhotoCount() - 1 << "): ";  
 cin >> index;  
 cin.ignore();  
   
 cout << "Enter tag to add: ";  
 getline(cin, tag);  
   
 if (gallery.addTagToPhoto(index, tag)) {  
 cout << "Tag added successfully." << endl;  
 } else {  
 cout << "Failed to add tag." << endl;  
 }  
 break;  
 }  
   
 case 6: {  
 int searchChoice;  
 cout << "\n=== Search Options ===" << endl;  
 cout << "1. Search by location" << endl;  
 cout << "2. Search by tag" << endl;  
 cout << "3. Search by date range" << endl;  
 cout << "4. Search by keyword prefix" << endl;  
 cout << "5. Search by description text" << endl;  
 cout << "Enter choice: ";  
 cin >> searchChoice;  
 cin.ignore();  
   
 Photo\* results[100];  
 int count = 0;  
   
 switch (searchChoice) {  
 case 1: {  
 string location;  
 cout << "Enter location to search: ";  
 getline(cin, location);  
   
 gallery.searchByLocation(location, results, count);  
 break;  
 }  
   
 case 2: {  
 string tag;  
 cout << "Enter tag to search: ";  
 getline(cin, tag);  
   
 gallery.searchByTag(tag, results, count);  
 break;  
 }  
   
 case 3: {  
 string startDateStr, endDateStr;  
 cout << "Enter start date (YYYY-MM-DD): ";  
 getline(cin, startDateStr);  
   
 cout << "Enter end date (YYYY-MM-DD): ";  
 getline(cin, endDateStr);  
   
 gallery.searchByDateRange(startDateStr, endDateStr, results, count);  
 break;  
 }  
   
 case 4: {  
 string prefix;  
 cout << "Enter keyword prefix to search: ";  
 getline(cin, prefix);  
   
 gallery.searchByPrefix(prefix, results, count);  
 break;  
 }  
   
 case 5: {  
 string text;  
 cout << "Enter text to search in descriptions: ";  
 getline(cin, text);  
   
 gallery.searchByDescription(text, results, count);  
 break;  
 }  
   
 default:  
 cout << "Invalid search option." << endl;  
 continue;  
 }  
   
 cout << "\nFound " << count << " results:" << endl;  
 for (int i = 0; i < count; i++) {  
 gallery.displayPhoto(results[i]);  
 }  
 break;  
 }  
   
 case 7: {  
 int sortChoice;  
 cout << "\n=== Sort Options ===" << endl;  
 cout << "1. Sort by date" << endl;  
 cout << "2. Sort by size" << endl;  
 cout << "3. Sort by popularity (view count)" << endl;  
 cout << "Enter choice: ";  
 cin >> sortChoice;  
 cin.ignore();  
   
 Photo\* results[100];  
   
 switch (sortChoice) {  
 case 1: {  
 char order;  
 cout << "Sort in descending order? (y/n): ";  
 cin >> order;  
 cin.ignore();  
   
 gallery.sortByDate(results, (order == 'y' || order == 'Y'));  
 break;  
 }  
   
 case 2: {  
 char order;  
 cout << "Sort in descending order? (y/n): ";  
 cin >> order;  
 cin.ignore();  
   
 gallery.sortBySize(results, (order == 'y' || order == 'Y'));  
 break;  
 }  
   
 case 3: {  
 char order;  
 cout << "Sort in descending order? (y/n): ";  
 cin >> order;  
 cin.ignore();  
   
 gallery.sortByPopularity(results, (order == 'y' || order == 'Y'));  
 break;  
 }  
   
 default:  
 cout << "Invalid sort option." << endl;  
 continue;  
 }  
   
 cout << "\nSorted Photos:" << endl;  
 for (int i = 0; i < gallery.getPhotoCount(); i++) {  
 cout << "[" << i << "] ";  
 gallery.displayPhoto(results[i]);  
 }  
 break;  
 }  
   
 case 8: {  
 int viewChoice;  
 cout << "\n=== View Options ===" << endl;  
 cout << "1. Most recent photos" << endl;  
 cout << "2. Most popular photos" << endl;  
 cout << "Enter choice: ";  
 cin >> viewChoice;  
 cin.ignore();

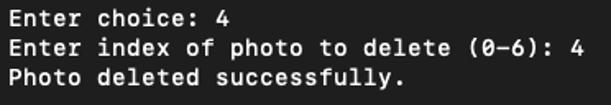
Photo\* results[10];  
 int count;  
   
 switch (viewChoice) {  
 case 1: {  
 gallery.getMostRecentPhotos(results, count);  
 cout << "\nMost Recent Photos:" << endl;  
 break;  
 }  
   
 case 2: {  
 gallery.getMostPopularPhotos(results, count);  
 cout << "\nMost Popular Photos:" << endl;  
 break;  
 }  
   
 default:  
 cout << "Invalid option." << endl;  
 continue;  
 }  
   
 for (int i = 0; i < count; i++) {  
 cout << "[" << i << "] ";  
 gallery.displayPhoto(results[i]);  
 }  
 break;  
 }  
   
 case 9: {  
 gallery.getDataStructureStats();  
 break;  
 }  
   
 default:  
 cout << "Invalid choice. Please try again." << endl;  
 }  
 }  
   
 cout << "Thank you for using Photo Gallery System!" << endl;  
 return 0;  
}

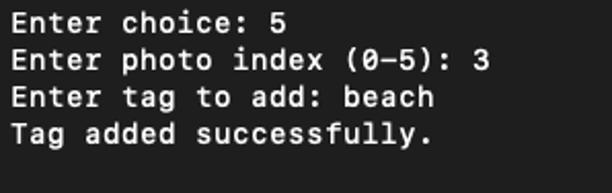
**Results & Output**

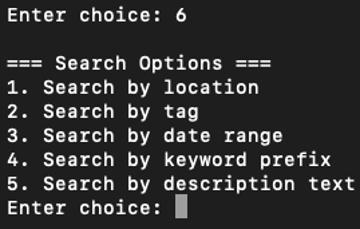


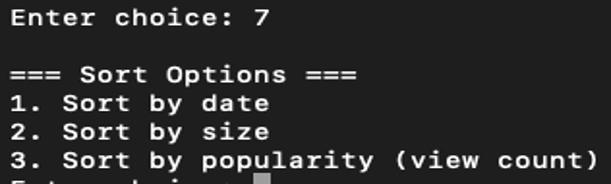


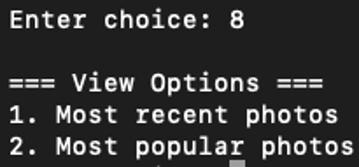


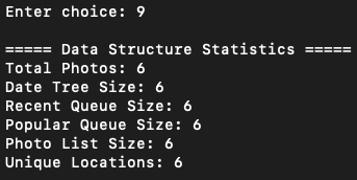














**Future Enhancements**

**1. Cloud Integration**

Future versions of the system can include seamless synchronization with cloud platforms such as Google Drive, Dropbox, or OneDrive. This would allow users to back up their photos securely, access them across devices, and restore them if needed—without compromising performance or privacy.

**2. Enhanced AI Models**

As machine learning technologies evolve, the system can integrate more powerful and efficient AI models. Improvements could include better accuracy in face detection, more context-aware tagging, and smarter caption generation that understands emotions, actions, or settings within images.

**3. User Collaboration**

To support families or teams, the system can be extended to include collaboration features. Users could co-manage shared albums, tag individuals together, and comment on or edit descriptions. This would foster a more interactive and social experience within private groups.

**4. Mobile Compatibility**

A responsive mobile application or web interface would allow users to manage their photo libraries anytime, anywhere. Whether it’s uploading photos directly from a phone or browsing albums on a tablet, mobile support will significantly improve accessibility and usage convenience.

**5. Voice Control**

Voice commands could make photo browsing even more natural. Integration with voice assistants like Google Assistant, Siri, or custom voice models would allow commands like:

* “Show me beach photos from 2022”
* “Find all photos with Dad”
* “Open my last vacation album”

This would particularly benefit users who prefer hands-free interaction.

**6. Augmented Reality (AR)**

AR can create immersive experiences with photo collections. For example:

* Users could view photo timelines as 3D floating albums.
* Memories could be anchored to physical locations using AR glasses or phones (e.g., "See all photos taken in Paris while standing there").
* Interactive photo walls could be created for events or storytelling.

This feature would bridge the gap between physical and digital memories in a futuristic, engaging way.