|  |
| --- |
| King Saud University  College of Computer and Information Sciences  Department of Computer Science  CSC 212 Data Structures Project Report – 2nd Semester 2024-2025  Developing a Photo Management Application |

**Authors**

|  |  |  |
| --- | --- | --- |
| Name | ID | List of all methods implemented by each student |
| Ragad Alrashid | **444200498** | **Album class + tag matching logic** |
| Munnerah Alhowekan | **444200541** | **Inverted index via BST (InvidexPhotoManager)** |
| Wajan Alhajiri | **444200867** | **Photo class + photoManager (basic)** |

# Introduction

This project focuses on building a **photo management system** that organizes photos based on their associated tags. The core idea is to **efficiently store, retrieve, and filter** photos based on multiple tag conditions. We implemented a traditional photo manager, and then enhanced it by introducing an **inverted index** using a **Binary Search Tree (BST)**, allowing faster multi-tag queries.  
This report presents the **specifications** of the classes and ADTs used, the **design** approach, **implementation** highlights, a **performance analysis** comparing traditional vs. inverted index search, and the final **conclusion** about the project work.

# Specification

### **LinkedList<T>**

* Stores elements in a sequence (one after another).
* **Main methods**:
  + insert(data): Add a new item at the start.
  + remove(data): Remove an item if it exists.
  + contains(data): Check if an item exists.
  + get(index): Get the item at a certain position.
  + size(): Get how many items are in the list.
  + copy(): Make a copy of the list.

### **BST<K, V>**

* A tree where each node has a key and a value.
* **Main methods**:
  + insert(key, value): Add a new key-value pair.
  + search(key): Find the value by its key.
  + delete(key): Remove a key-value pair.

### **Photo**

* Represents a single photo.
* **Data**:
  + path: The file path of the photo.
  + tags: A list of tags (keywords) describing the photo.
* **Main methods**:
  + getPath(): Return the photo's path.
  + getTags(): Return a list of the photo's tags.

### **PhotoManager**

* Manages a collection of photos.
* **Main methods**:
  + addPhoto(p): Add a new photo.
  + deletePhoto(path): Remove a photo by its path.
  + getPhotos(): Return all photos.

### **InvIndexPhotoManager (extends PhotoManager)**

* Adds a fast search feature using an inverted index (tags → photos).
* **Extra data**:
  + invertedIndex: A BST that maps each tag to a list of photos with that tag.
* **Extra methods**:
  + getPhotosByCondition(condition): Find photos that match a set of tags.

### **Album**

* Represents a collection of photos based on a search condition.
* **Data**:
  + name: Album name.
  + condition: Search rule (example: "beach AND sunset").
  + manager: Reference to PhotoManager or InvIndexPhotoManager.
* **Main methods**:
  + getPhotos(): Get photos matching the condition.
  + getNbComps(): Get the number of comparisons made during the search.

# Design

The system design consists of **two layers**:

* **Basic Photo Management Layer** (PhotoManager): Handles basic add, delete, and retrieve operations using a simple list.
* **Advanced Search Layer** (InvIndexPhotoManager): Enhances search using an **inverted index** (BST), mapping each tag to a list of photos that contain it.

|  |  |
| --- | --- |
| **Class** | **Depends on** |
| PhotoManager | Photo, LinkedList |
| InvIndexPhotoManager | PhotoManager, BST, LinkedList |
| Album | PhotoManager (or InvIndexPhotoManager) |
| BST | Generic data types |
| LinkedList | Generic data types |

# Implementation

# **Inverted Index Construction:**

# On adding a photo, each of its tags is inserted into a BST where the tag is the key and the value is a list of photos with that tag.

# If the tag does not exist, a new entry is created.

LinkedList<String> tags = p.getTags();

for (int i = 0; i < tags.size(); i++) {

String tag = tags.get(i);

LinkedList<Photo> photos = invertedIndex.search(tag);

if (photos == null) {

photos = new LinkedList<>();

invertedIndex.insert(tag, photos);

}

photos.insert(p);

}

* **Photo Deletion**:
  + When a photo is deleted, it is removed from the global photo list **and** from all corresponding tag lists in the inverted index.
* **Optimized Search**:
  + To search for photos matching multiple tags, the algorithm first finds the **smallest tag list** and then checks each photo individually.

for (String tag : tags) {

LinkedList<Photo> photos = invertedIndex.search(tag);

if (photos.size() < smallestSize) {

smallestSize = photos.size();

smallestList = photos;

}

}

* **Performance Counting**:
  + In Album, the number of comparisons (nbComps) is counted manually depending on the search strategy used.

# Performance analysis

|  |  |
| --- | --- |
| **Scenario** | **Time Complexity (Big-O)** |
| **Album.getPhotos() without inverted index** | **O(n \* m)** |
| **Album.getPhotos() with inverted index** | **O(k + m')** |

Where:

* n = number of all photos
* m = number of tags per photo
* k = number of tags in the search condition
* m' = size of the smallest matching tag list

Before using Inverted Index:

Album had to check every photo against every tag ⇒ O(n \* m).

After using Inverted Index:

Search is faster:

Find tag lists ⇒ O(log t) per tag, where t = number of tags in BST.

Iterate over the smallest list and validate ⇒ O(m') comparisons.

Resulting in much faster retrieval, especially when the database grows.

# Conclusion

In this project, we successfully designed and implemented a **photo management system** with an **optimized multi-tag search** using an **inverted index based on BST**.  
 We demonstrated how the system scales much better after optimization, reducing search time significantly.  
 The modular class structure ensures **easy extensions** (such as adding OR queries, NOT queries, etc.) in the future.

**Limitations**:

* The BST is not balanced, so in worst cases (e.g., tags inserted in sorted order), search may degrade to O(n).
* Tags must exactly match; no partial matching or case insensitivity yet.

**Future Work**:

* Implement a **self-balancing BST** (like AVL Tree).
* Allow **partial matches** or **fuzzy search**.
* Support advanced queries like **OR**, **NOT**, and parentheses in conditions.