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
Haifa Alsaif	443202006	Implemented Photo, LinkedList, BST, and report.
Raghad Alhulwah	444200453	Implemented Album and report.
Juri Alghamdi	444201188	Implemented PhotoManager, InvIndexPhotoManager, and report.

Supervised By: *Dr. Noof Alfear*

1. Introduction

In this project, we designed and implemented a Photo Management System using fundamental data structures where each photo is associated with a set of descriptive tags and albums are dynamically created based on logical conditions over these tags, such as “animal AND grass,” to organize photos efficiently.

To organize and retrieve photos, two main approaches were implemented:

- Using a simple **Linked List** data structure.
- Using an Inverted Index built on a **Binary Search Tree (BST)** for optimized searching.

The objective of this report is to describe the system’s specifications, design decisions, key implementation details, and perform a theoretical performance analysis (Big-O) of the Album.getPhotos() method before and after using the inverted index.

2. Specification

Class Photo:

Elements:
The elements are of type String (Tags associated with the photo are stored in a LinkedList<String>).

Structure:
The Photo class contains a String for the photo's path and a LinkedList<String> for storing all the tags associated with that photo.

Domain:
The number of elements in the list of tags is finite and depends on the number of tags associated with the photo. The type name of elements in the domain is String.

<u>**Operations:**</u> We assume all operations operate on a Photo object.

1. **Constructor:** Photo(String path, LinkedList<String> tags)

- **Requires:** None
- **Input:** A String path representing the photo's file path and a LinkedList<String> tags representing the tags associated with the photo.

- **Results:** Initializes the photo object with the given path and tags. Tags are copied into the allTags linked list.
- **Output:** None.

2. **Method:** getPath()

- **Requires:** None
- **Input:** None
- **Results:** Returns the path (full file name) of the photo, which is used as a unique identifier for the photo.
- **Output:** String (the file path of the photo).

3. Method: getTags()

- **Requires:** None
- **Input:** None
- **Results:** Returns a new LinkedList<String> containing the tags associated with the photo.
- **Output:** LinkedList<String> (a list of tags).

4. **Method:** toString()

- **Requires:** None
- **Input:** None
- **Results:** Returns a String representation of the Photo object, including the path and all associated tags.
- **Output:** String (formatted string representing the photo).

Class PhotoManager:

Elements:
The elements are of type Photo, and these elements are stored in a LinkedList<Photo>. Each Photo contains data related to the photo (path and associated tags).

Structure:
The PhotoManager class maintains a linked list (LinkedList<Photo>) of photos which is A linked list that stores the photos managed by this class.

Domain:
The number of elements (photos) in the list is finite, bounded by the available memory. The type of elements in the domain is Photo.

Operations: We assume all operations operate on a PhotoManager object.

1. **Constructor:** PhotoManager()

- **Requires:** None
- **Input:** None
- **Results:** Initializes an empty LinkedList<Photo> to store photos.
- **Output:** None.

2. **Method:** addPhoto(Photo p)

- **Requires:** The photo p does not already exist in the list (determined by its path).
- **Input:** A Photo object p.
- **Results:** Adds the photo p to the photos list if the photo with the same path is not already present.
- **Output:** None.

3. **Method:** IsPhototAvailable(String path, LinkedList<Photo> list)

- **Requires:** None
- **Input:** A String path representing the path of the photo to check. A LinkedList<Photo> named list to search for the photo.
- **Results:** Returns true if the photo with the specified path exists in the list. Otherwise, returns false.
- **Output:** boolean.

4. **Method:** deletePhoto(String path)

- **Requires:** The photo with the given path must exist in the list.
- **Input:** A String path representing the path of the photo to be deleted.
- **Results:** Deletes the photo with the specified path from the photos list if it exists.
- **Output:** None.

5. **Method:** getPhotos()

- **Requires:** None
- **Input:** None
- **Results:** Returns the LinkedList<Photo> that contains all the photos managed by the PhotoManager.
- **Output:** LinkedList<Photo>.

Class BST:

Elements:
The elements are of generic type <T>, stored in nodes of the binary search tree. Each node contains a key (String key) and associated data (T data).The tree structure consists of nodes

connected in a binary search tree manner, where each node has at most two children: left and right.

Structure:
The elements are arranged in a binary search tree structure. The root node is the starting point, and each node has at most two child nodes: left and right. The first node is called the root, and there is a current node used for various operations such as searching, inserting, and updating.

Domain:
The number of elements in the tree is bounded by the size of the tree, which grows or shrinks based on insertions and deletions. The domain is finite, with each element in the domain being of type <T>.Type name of elements in the domain BSTNode<T>, where each node contains a key and associated data of type <T>.

Operations:

1. **Method:** empty( )

   - **Requires:** None.
   - **Input:** None.
   - **Results:** Returns true if the tree is empty, false otherwise.
   - **Output:** boolean.

2. **Method:** full( )

   - **Requires:** None.
   - **Input:** None.
   - **Results:** Always returns false because the tree is not bounded in size.
   - **Output:** boolean.

3. **Method:** retrieve( )

   - **Requires:** The current node is set.
   - **Input:** None.
   - **Results:** Returns the data stored in the current node.
   - **Output:** T data.

4. **Method:** findkey(String tkey)

   - **Requires:** The tree is not empty.
   - **Input:** tkey (key to search for in the tree).
   - **Results:** If the key is found, sets current to the corresponding node and returns true. If not, returns false.
   - **Output:** boolean.

5. **Method:** insert(String k, T val)

- **Requires:** The tree is not full (there is no size limitation).
- **Input:** k (key) and val (data to be inserted).
- **Results:** Inserts the new node with key k and data val into the tree if the key does not already exist. Returns true if insertion is successful, false if the key already exists.
- **Output:** boolean.

6. **Method:** remove_key(String tkey)

- **Requires:** The tree contains the key tkey.
- **Input:** tkey (key to remove from the tree).
- **Results:** Removes the node with key tkey from the tree. Returns true if the key was removed, false otherwise.
- **Output:** boolean.

7. **Method:** remove_aux (String key, BSTNode<T> p, boolean flag)

- **Requires:** A node p in the tree.
- **Input:** key (key to remove), p (current node), flag (indicates whether a node was removed).
- **Results:** Recursively removes a node with the specified key and adjusts the tree structure.
- **Output:** BSTNode<T> (the updated node or subtree).

8. **Method:** find_min (BSTNode<T> p)

- **Requires:** A non-null node p.
- **Input:** p (node from which to find the minimum).
- **Results:** Returns the node with the smallest key in the subtree rooted at p.
- **Output:** BSTNode<T> (the node with the minimum key).

9. **Method:** update (String key, T data)

- **Requires:** The tree contains the key.
- **Input:** key (key to update) and data (new data).
- **Results:** Removes the existing node with the specified key and inserts a new node with the updated data.
- **Output:** boolean (indicating whether the update was successful).

10. **Method:** inOrder ( )

- **Requires:** The tree is not empty.
- **Input:** None.
- **Results:** Returns a string containing the keys of the nodes in the tree, in ascending order.
- **Output:** String All Keys (keys in ascending order).

11. **Method:** inorder (BSTNode<T> p)

- **Requires:** A non-null node p.

- **Input:** p (current node in the tree).
- **Results:** Recursively traverses the tree in an in-order fashion and concatenates the keys into the AllKeys string.
- **Output:** None.

Class LinkedList:

Elements:
The elements are of generic type T. These elements are stored in nodes of type Node<T>, which are linked to each other to form the list.

Structure:
The LinkedList class is composed of nodes (Node<T>) that store elements of type T. Each node has a reference to the next node. Head Points to the first node in the list. current Points to the current node (for traversal purposes). size: Tracks the number of elements in the list.

Domain:
The number of elements in the list is finite and bounded by the available memory. The type name of elements in the domain is T.

Operations: We assume all operations operate on a LinkedList<T> object.

1. **Method:** Constructor (LinkedList())

- **Requires:** None
- **Input:** None
- **Results:** Initializes an empty linked list.
- **Output:** None.

2. **Method:** empty()

- **Requires:** None
- **Input:** None
- **Results:** Returns true if the list is empty; otherwise, returns false.
- **Output:** boolean.

3. **Method:** last()

- **Requires:** None
- **Input:** None
- **Results:** Returns true if the current node is the last node in the list; otherwise, returns false.
- **Output:** boolean.

4. **Method:** full()

- **Requires:** None
- **Input:** None
- **Results:** Always returns false (the linked list is never full, assuming no memory constraints).
- **Output:** boolean.

5. **Method:** findFirst()

- **Requires:** None
- **Input:** None
- **Results:** Sets the current node to the first node in the list (head).
- **Output:** None.

6. **Method:** findNext()

- **Requires:** None
- **Input:** None
- **Results:** Sets the current node to the next node in the list.
- **Output:** None.

7. **Method:** retrieve()

- **Requires:** The current node must not be null.
- **Input:** None
- **Results:** Returns the data of the current node.
- **Output:** T (the data stored in the current node).

8. **Method:** update(T val)

- **Requires:** The current node must not be null.
- **Input:** A value val of type T to update the data in the current node.
- **Results:** Updates the data of the current node with the provided value.
- **Output:** None.

9. **Method:** insert(T val)

- **Requires:** None
- **Input:** A value val of type T to insert into the list.
- **Results:** Inserts the given value at the position of the current node. If the list is empty, the node is added as the first node.
- **Output:** None.

10. **Method:** remove()

- **Requires:** The current node must not be null.
- **Input:** None
- **Results:** Removes the current node from the list. If the current node is the first node (head), the head is updated.
- **Output:** None.

11. **Method:** getSize()

- **Requires:** None
- **Input:** None
- **Results:** Returns the number of elements in the linked list.
- **Output:** int (the size of the list).

Class Album:

Elements:
The elements are of generic type Photo (The elements are stored in a LinkedList<Photo> or BST<LinkedList<Photo>>).

Structure:
The elements are linearly arranged. The first element is referred to as the head. There is an element called current, which represents the current photo.

Domain:
The number of elements is finite, based on the photos in the album, and the type name of elements in the domain is Photo.

Operations: We assume all operations operate on an Album.

Constructor Album (String name, String condition, PhotoManager manager)

- **requires:** valid name, condition, and a non-null PhotoManager.
- **input:** album name, condition, manager.
- **results:** constructs a new album with the given attributes; initializes NbComps to 0.
- **output:** a new Album object.

1. **Method:** getName()

- **Requires:** None.
- **Input:** None.
- **Results:** Returns the name of the album.
- **Output:** String (name of the album).

2. **Method:** getCondition()

- **Requires:** None
- **Input:** None
- **Results:** Returns the condition associated with the album.
- **Output:** String (album condition).

3. **Method:** getManager()

- **Requires:** None
- **Input:** None
- **Results:** Returns the PhotoManager object associated with the album.
- **Output:** PhotoManager (manager).

4. **Method:** getInvManager()

- **Requires:** None
- **Input:** None
- **Results:** Returns the InvIndexPhotoManager object associated with the album.
- **Output:** InvIndexPhotoManager (invmanager).

5. **Method:** getNbComps()

- **Requires:** None
- **Input:** None
- **Results:** Returns the number of tag comparisons used to find all photos of the album.
- **Output:** int (number of comparisons).

6. **Method:** getPhotos()

- **Requires:** None
- **Input:** None
- **Results:** Returns all photos that satisfy the album condition. The result is based on the user's choice between LinkedList or BST.
- **Output:** LinkedList<Photo> (filtered photos list).

7. **Method:** getPhotosLL()

- **Requires:** None
- **Input:** None
- **Results:** Returns a filtered list of photos using the LinkedList structure based on the album's condition.
- **Output:** LinkedList<Photo> (filtered photos list using LinkedList).

8. **Method:** getPhotosBST()

- **Requires:** None
- **Input:** None
- **Results:** Returns a filtered list of photos using the BST structure based on the album's condition.
- **Output:** LinkedList<Photo> (filtered photos list using BST).

9. **Method:** allAvilable()

- **Requires:** None
- **Input:** LinkedList<String> AllTags, String[] Array
- **Results:** Returns true if all the tags in the condition are found in the photo tags.
- **Output:** boolean (true or false).

10. **Method:** Function()

- **Requires:** None
- **Input:** LinkedList<Photo> list1, LinkedList<Photo> list2
- **Results:** Returns a list of photos that exist in both list1 and list2.
- **Output:** LinkedList<Photo> (common photos between two lists).

11. **Method:** menu()

- **Requires:** None
- **Input:** None
- **Results:** Prompts the user to select between LinkedList or BST for photo retrieval.
- **Output:** int (user's choice).

Class InvIndexPhotoManager:

Elements:
The elements are of generic type Photo (Stored in a BST<LinkedList<Photo>> structure, where each key represents a tag and the value is a list of photos associated with that tag).

Structure:
The elements are arranged in a Binary Search Tree (BST). The keys are tags (String), and the values are LinkedList<Photo> objects containing the photos related to the tag.

Domain:
The number of elements is finite and depends on the number of tags and photos in the album. The type name of elements in the domain is LinkedList<Photo>.

Operations: We assume all operations operate on an InvIndexPhotoManager.

1. **Method:** addPhoto(Photo p)

- **Requires:** None
- **Input:** A Photo object p to be added.
- **Results:** Adds the given photo to the inverted index. It adds the photo under the key " " (empty string) and under each of the tags associated with the photo.
- **Output:** None.

2. **Method:** deletePhoto(String path)

- **Requires:** The photo with the given path exists in the inverted index.
- **Input:** A String path representing the file path of the photo to be deleted.
- **Results:** Deletes the photo from the inverted index, removing it from the lists of photos associated with its tags. If any tag no longer has associated photos, the tag is removed from the index.
- **Output:** None.

3. **Method:** getPhotos()

- **Requires:** None
- **Input:** None
- **Results:** Returns the inverted index, which is a BST<LinkedList<Photo>> containing the tags and the associated list of photos for each tag.

- **Output:** BST<LinkedList<Photo>> (the inverted index).

Class Test:

Elements:

The elements are objects of type InvIndexPhotoManager, PhotoManager, Photo, and Album. Also uses LinkedList<String> and LinkedList<Photo> internally for tags and photo lists.

Structure:

The objects are created and organized sequentially.Photos are added first, then albums are created based on certain conditions .There is a sequence of method calls to display or modify the data.

Domain:

The number of photos, albums, and tags is finite and defined manually in the program.Type name of elements in the domain: Test

3. ADT Test: Specification

Operations:
We assume all operations operate within the main method of Test.

1. **Method:** main(String[] args)
    - **requires:** None
    - **input:** None (data hardcoded inside the method).
    - **results:**
        - Creates managers for photos.
        - Adds photos with predefined tags.
        - Creates albums based on tag conditions.
        - Prints photo paths and their tags.
        - Prints albums' names, conditions, and contained photos.
        - Deletes a photo and updates albums accordingly.
    - **output:** Display information to console.
2. **Method: toTagsLinkedList(String tags)**
    - **requires:** Non-null comma-separated String of tags.
    - **input:** A string containing tags separated by commas.
    - **results:**
        - Splits the string into individual tags.
        - Inserts each tag into a LinkedList.
    - **output:** A LinkedList<String> containing all tags.
3. **Method :printLL(LinkedList<String> list)**
    - **requires:** Non-null LinkedList.
    - **input:** A LinkedList<String>.
    - **results:**
        - Iterates through the list and prints each tag.
    - **output:** Printed tags to console.
4. **Method: printLLPhoto(LinkedList<Photo> list)**
    - **requires:** Non-null LinkedList.
    - **input:** A LinkedList<Photo>.
    - **results:**
        - Iterates through the list and prints the path of each photo.
    - **output:** Printed photo paths to console.

4. Design

The system is designed to efficiently manage photo collections using custom-built data structures. The main goal is to store photos with tags and create albums based on tag-based conditions. The design ensures simplicity, efficiency, and clear responsibilities across different classes.

Class	Description
Photo	Represents an individual photo, storing its file path and associated tags.
PhotoManager	Uses `LinkedList<Photo>` to manage a sequential collection of photos, allowing add, delete, and retrieve operations.
InvIndexPhotoManager	Manages an inverted index on top of the BST, where each tag points to a list of photos, enabling faster search operations.
Album	Represents a photo album defined by a condition (logical combination of tags) and retrieves photos based on either: - PhotoManager (sequential search using LinkedList). - InvIndexPhotoManager(efficient search using BST).
LinkedList	Provides a generic implementation of a singly linked list to manage both photos and tags.
BST	Implements a binary search tree (BST) to associate each tag with a `LinkedList<Photo>` containing all photos with that tag.

This design ensures that the system supports two types of photo retrieval:

- A sequential search through a linked list for straightforward access.
- An optimized indexed search through a binary search tree for faster performance in large datasets.

Thus, the interaction between these components achieves a balance between simplicity and efficiency, adapting to the requirements of different scenarios.

5. Implementation

This section outlines the key components of the photo management system. The system is built around several classes that interact to manage and retrieve photos based on specific conditions, such as tags. The main classes involved are Album, InvIndexPhotoManager, PhotoManager, Photo, LinkedList, and BST.

- **Album Class**

The Album class represents a photo album and is responsible for retrieving photos based on a specified condition, which could be a tag or a combination of tags. It can use two different methods for retrieval:

- A basic method that uses a simple PhotoManager class, which stores photos in a basic list and retrieves them by iterating over the list.
- An optimized method that uses the InvIndexPhotoManager class, which stores photos in an inverted index (a data structure that maps tags to lists of photos). This method enables faster photo retrieval, especially for large datasets, by utilizing a binary search tree (BST).

- **InvIndexPhotoManager Class**

The InvIndexPhotoManager class is responsible for maintaining an inverted index of photos. This index maps tags to lists of photos that have those tags. The class uses a binary search tree (BST) to enable fast lookups for photos based on tags. When a photo is added or removed, the index is updated accordingly to maintain efficiency in photo retrieval.

- **Photo Class**

The Photo class represents an individual photo in the system. Each photo has a unique identifier, its file path, and a set of associated tags. The tags are used to filter and categorize photos, making it easier to search for specific photos in an album.

- **PhotoManager Class**

The PhotoManager class manages a collection of photos in a simple linked list. It provides methods for adding new photos, deleting photos by path, and retrieving all stored photos. While this approach is simpler, it can be inefficient when retrieving photos based on specific tags, as it requires checking each photo one by one.

- **LinkedList Class**

The LinkedList class is a custom implementation of a singly linked list, used to store photos, tags, or other elements within the system. It provides operations such as inserting, retrieving, and removing elements. This data structure is chosen for its flexibility and ease of use when managing collections of items in the system.

- **BST Class**

The BST class implements a binary search tree to manage tags and their associated photos. Each node in the tree holds a tag and a linked list of photos that share that tag. The tree allows for efficient tag lookups, insertions, and deletions.

- **Insertion**: Tags are added in alphabetical order. If a tag already exists, new photos are appended to its list.
- **Searching**: Tags can be searched with a time complexity of O(log n), making retrieval faster than a linear search.
- **Deletion**: Tags can be removed along with their photos, ensuring the tree remains balanced.

6. Performance analysis

The performance of the Album.getPhotos() method varies depending on the data structure used for photo retrieval:

5.1 Before Using Inverted Index (Using LinkedList)

When using the PhotoManager class (which stores photos sequentially in a LinkedList), the method getPhotosLL() works as follows:

- It iterates over all the photos in the LinkedList.
- For each photo, it checks all tags against the album condition (which can consist of one or more tags).
- This checking is done linearly through the list of tags for each photo.

Thus, the time complexity depends on:

- n: the number of photos.
- m: the number of tags associated with each photo (average).
- k: the number of tags in the album condition.

Time Complexity (Big-O):

O(n \times m \times k)

- n for iterating over all photos,
- m for iterating over all tags of a photo,
- k for checking the condition (AND combination of k tags).

In the worst case, all photos must be checked, and for each photo, all tags must be compared with all condition tags.

5.2 After Using Inverted Index (Using BST)

When using the InvIndexPhotoManager class (which maintains an inverted index with a Binary Search Tree (BST)), the method getPhotosBST() works as follows:

- The album condition is split into its tags.
- For each tag, it searches the BST to find a list of photos associated with that tag.
- The BST search operation has $O(\log t)$ time complexity, where t is the number of unique tags in the system.
- After finding photos for each tag, it intersects the lists to find the common photos.

Thus, the time complexity depends on:

- k: the number of tags in the album condition.
- p: the average number of photos per tag.

Time Complexity (Big-O):

$O(k \times (\log t + p))$

- $k \times \log t$ for searching k tags in the BST,
- $k \times p$ for processing the intersection of photo lists.

Notes:

- Since search in a BST is faster than linear scanning, and if the number of photos per tag p is small compared to the total number of photos n, the inverted index greatly reduces the time needed for retrieval.
- This optimization is especially beneficial as the number of photos grows large.

5.3 Summary

Comparing both methods, using the inverted index with a BST significantly improves retrieval efficiency compared to a sequential LinkedList search.

Without the inverted index, the time complexity is $O(n \times m \times k)$ due to scanning all photos and tags.

With the inverted index, the complexity is reduced to $O(k \times (\log t + p))$, allowing faster lookups and intersections.

Thus, the inverted index greatly enhances performance, especially when dealing with large photo collections and complex tag conditions.

7. Conclusion

In this project, we designed and implemented a photo album management system focused on efficiently organizing a collection of photos based on tags. We started by creating a basic Album class, allowing users to create albums based on conditions (tags) and retrieve matching photos.

Initially, the retrieval process was simple but inefficient, with a time complexity of O(n), where n is the number of photos.

To improve performance, we introduced an inverted index. This optimization used a Binary Search Tree (BST) to store and quickly retrieve tags, reducing the time complexity for tag retrieval to O(log n), where n is the number of unique tags. The optimized system showed significant performance improvements, especially for larger datasets.

While the system works well, there are areas for improvement:

1. **Scalability**: The system may face challenges with very large datasets, particularly with memory usage.
2. **Tagging System**: The current system supports basic tag searches, but more advanced search features could be added.

For future work, we could:

1. Implement a database to manage data more efficiently.
2. Explore additional techniques to enhance performance with larger datasets.

In conclusion, the project successfully improved photo retrieval speed using the inverted index and provides a solid foundation for future enhancements.

8. GitHub link:

photo-management-datastructures