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
Amjad Yahya Almalki	444200474	Photo, LinkedList, BST, InvIndexPhotoManager, report
Shahad Homoud Alenizi	444201038	Album, Test_main, Test, report
Leen Ahmad Sewar	444204749	PhotoManager, InvAlbum, report

1. Introduction

In this project, we developed a Photo Management Application that allows users to efficiently organize and retrieve photos based on descriptive tags. Each photo is associated with multiple tags that describe its content, and users can create albums by specifying logical conditions on these tags.

The project is divided into two parts:

- In the first phase, we implemented the basic functionality using some data structures such as a singly linked list. Photos are manually searched based on their tags when creating albums.
- In the second phase, we optimized the search process by introducing an Inverted Index stored inside a Binary Search Tree (BST). This structure allows faster access to photos based on tags, improving the performance significantly when dealing with large numbers of photos and complex queries.

This report documents the design, specifications, implementation details, and theoretical performance analysis of both phases of the project.

2. Specification

Class Photo:

Attributes:

path: (String) Represents the full file name (path) of the photo.

photoTags: (LinkedList) Stores the tags associated with the photo.

Method:

Photo(String path, LinkedList<String> tags): Initializes a photo with a path and copies its tags into the photoTags list.

getPath(): Retrieve the path of the photo.

getTags(): Retrieve a copy of the tags associated with the photo.

toString(): Return a string representation of the photo information.

Class PhotoManager:

Attributes:

PhotosList : LinkedList of Photo objects that stores all the managed photos.

Constructor:

PhotoManager() : Initializes an empty linked list to store photos.

Methods:

getPhotos() : Returns the list of managed photos.

PhotoAvailable(String path, LinkedList<Photo> PhotoList) : Checks if a photo with the given path exists in the list.

addPhoto(Photo p) : Adds a photo to the list if it does not already exist.

deletePhoto(String path) : Deletes a photo with the given path from the list.

Class LinkedList:

Attributes:

head: (Node) Points to the first node in the list.

current: (Node) Points to the current node during traversal.

size: (int) Tracks the number of elements in the list.

Methods:

LinkedList(): Initializes an empty linked list.

insert(T val): Insert an element after the current node and move the current pointer to it.

remove(): Remove the current element.

findFirst(): Move the current pointer to the first node.

findNext(): Move the current pointer to the next node.

retrieve(): Retrieve the element at the current node.

empty(): Check if the list is empty.

last(): Check if the current node is the last node.

getSize(): Return the number of elements in the list.

Class BST:

Attributes:

root: (BSTNode) The root node of the binary search tree.

current: (BSTNode) A pointer to the current node used during operations.

size: (int) The number of nodes in the tree.

AllKeys: (String) A string that stores all keys during in-order traversal.

Method:

insert(String key, T val): Insert a key-value pair while maintaining the BST properties.

removeKey(String key): Remove the node containing the specified key.

findKey(String key): Search for a node with the specified key.

retrieve(): Retrieve the value associated with the current node.

empty(): Check if the BST is empty.

inOrder(): Traverse the BST and return all keys sorted in ascending order.

Class InvIndexPhotoManager:

Attributes:

TagIndex: (BST<LinkedList>) A binary search tree where each key is a tag and each value is a linked list of photos containing that tag.

Method:

addPhoto(Photo p): Insert each tag of the given photo into the BST. If a tag already exists, append the photo to its list; otherwise, create a new entry.

deletePhoto(String path): For each tag in the photo, remove the photo from the corresponding list in the BST.

Class Album:

Attributes:

private String name; Represents the name of the album.

private String condition; Represents the condition that photos must satisfy to be part of the album (e.g., tags combined by AND).

private PhotoManager manager; The photo manager that holds the list of available photos.

private int NbComps; Tracks the number of tag comparisons made when retrieving photos.

Methods:

public Album(String name, String condition, PhotoManager manager). Constructor to create an Album by initializing its name, condition, manager, and setting NbComps to 0.

public String getName() Returns the name of the album.

public String getCondition(). Returns the condition associated with the album.

public PhotoManager getManager(). Returns the PhotoManager linked to the album.

public int getNbComps(). Returns the number of tag comparisons made during photo retrieval.

public LinkedList<Photo> getPhotos(). Retrieves all photos from the manager that satisfy the album condition.

private boolean allAvailable(LinkedList<String> AllTags, String[] arr). Helper method that checks whether all required tags are available in a photo's tag list.

Class InvAlbum:

Attributes:

String name : stores the name of the album.

String condition : stores the condition to retrieve photos.

InvIndexPhotoManager invmanager : manages the inverted index of photos.

int NbComps : counts the number of tag comparisons.

Methods:

InvAlbum(String name, String condition, InvIndexPhotoManager manager) .Constructor. Initializes the album with a name, a condition, and a photo manager. Initializes NbComps to 0.

String getName() .Returns the name of the album.

String getCondition() .Returns the condition of the album.

InvIndexPhotoManager getManager() .Returns the manager associated with the album.

int getNbComps() .Returns the number of tag comparisons performed.

LinkedList<Photo> getPhotos() Returns all photos that satisfy the album condition. If the condition is empty, retrieves all photos. Otherwise, searches for photos that match the tags using the inverted index. Utilizes private helper methods Fun1 and Fun2 to combine results according to "AND" or "OR" operations.

private LinkedList<Photo> Fun1(LinkedList<Photo> L1, LinkedList<Photo> L2) . Handles "AND" operation between two lists: retrieves photos that exist in both L1 and L2.

private LinkedList<Photo> Fun2(LinkedList<Photo> L1, LinkedList<Photo> L2) .Handles "OR" operation between two lists: retrieves photos that exist in either L1 or L2 without duplication.

3. Design

This project follows object-oriented programming principles, ensuring each class has a distinct responsibility:
• Photo objects are responsible for holding image paths and associated tags.
• PhotoManager handles fundamental photo management tasks.
• Album generates dynamic collections of photos based on AND conditions across tags.
• InvIndexPhotoManager enhances search performance using an Inverted Index built with Binary Search Trees (BSTs).

In the optimized design, searches no longer require scanning all photos, significantly improving efficiency.

4. Implementation

4.1 LinkedList<T> Class

Data Representation

The linked list is represented using two pointers:

- head: a reference to the first node in the list.
- current: a reference to the current node used for traversing or updating the list. Representation

Each node (Node<T>) contains:

- data: the actual element stored.
- next: a pointer to the next node.

The list structure is a singly linked chain of nodes:

head → Node(data1, next) → Node(data2, next) → ... → Node(dataN, null)

Operations Implementation

- findFirst()
  Sets the current pointer to the first element:

```
public void findFirst() {
   current = head;
}
```

- findNext()
  Moves the current pointer to the next node:

```
public void findNext() {
   current = current.next;
}
```

- retrieve()
  Retrieves the data from the current node:

```
public T retrieve() {
   if (current == null) return null;
   return current.data;
}
```

- update(T val)
  Updates the data of the current node:

```
public void update(T val) {
   current.data = val;
}
```

- insert(T val)
  Inserts a new node after the current node:

```
public void insert(T val) {
   Node<T> tmp;
   if (empty()) {
      current = head = new Node<T>(val);
   } else {
      tmp = current.next;
      current.next = new Node<T>(val);
      current = current.next;
      current.next = tmp;
   }
   size++;
}
```

- remove()
  Removes the current node:

```
public void remove() {
   if (current == head) {
      head = head.next;
   } else {
      Node<T> tmp = head;
      while (tmp.next != current)
         tmp = tmp.next;
      tmp.next = current.next;
   }
   if (current.next == null)
      current = head;
   else
      current = current.next;
   size--;
}
```

- empty()
  Checks if the list is empty:

```
public boolean empty() {
   return head == null;
}
```

- last()
  Checks if the current node is the last:

```
public boolean last() {
   if (current == null) return true;
   return current.next == null;
}
```

- full()
  Always returns false (linked list is dynamic):

```
public boolean full() {
   return false;
}
```

4.2 Photo Class

Data Representation

The Photo class consists of:
- path: a String representing the full file path of the photo.
- photoTags: a LinkedList<String> containing the tags associated with the photo.

Operations Implementation

Constructor

- Deep copies the provided list of tags:

```
if (!tags.empty()) {
   tags.findFirst();
   do {
      photoTags.insert(tags.retrieve());
      if (tags.last()) break;
      tags.findNext();
   } while (true);
}
```

- getPath()
  Returns the path of the photo:

```
public String getPath() {
   return path;
}
```

- getTags()
  Returns the list of tags:

```
public LinkedList<String> getTags() {
   return photoTags;
}
```

4.3 PhotoManager Class

Data Representation

The PhotoManager class maintains:

- PhotosList: a LinkedList<Photo> containing all managed photos.

Operations Implementation

- addPhoto(Photo p)

Adds a photo if it does not already exist in the list:

```
public void addPhoto(Photo p) {

   if (!PhotoAvailable(p.getPath(), PhotosList)) {

      PhotosList.insert(p);

   }

}
```

- deletePhoto(String path)

Deletes a photo by matching its path:

```
public void deletePhoto(String path) {

   if (!PhotoAvailable(path, PhotosList)) return;

   if (!PhotosList.empty()) {

      boolean exist = false;
```

```
        PhotosList.findFirst();
        while (!exist && !PhotosList.last()) {
            Photo photo = PhotosList.retrieve();
            if (photo.getPath().compareToIgnoreCase(path) == 0) {
                exist = true;
                PhotosList.remove();
            }
            PhotosList.findNext();
        }
        if (!exist) {
            Photo photo = PhotosList.retrieve();
            if (photo.getPath().compareToIgnoreCase(path) == 0) {
                PhotosList.remove();
            }
        }
    }
}
```

- PhotoAvailable(String path, LinkedList<Photo> PhotoList)

Checks whether a photo with a specific path exists:

```
private boolean PhotoAvailable(String path, LinkedList<Photo> PhotoList) {
    if (PhotoList.empty()) return false;
    PhotoList.findFirst();
    while (!PhotoList.last()) {
        if (PhotoList.retrieve().getPath().compareToIgnoreCase(path) == 0)
            return true;
        PhotoList.findNext();
    }
    return PhotoList.retrieve().getPath().compareToIgnoreCase(path) == 0;
}
```

4.4 Album Class

Data Representation

The Album class contains:
- name: the album's name.
- condition: a string describing the tag condition (tags separated by "AND").
- manager: a reference to the PhotoManager.
- NbComps: an integer counting the number of tag comparisons performed.

Operations Implementation

- getPhotos()
  Retrieves a list of photos that satisfy the album's condition:

```
public LinkedList<Photo> getPhotos() {
    LinkedList<Photo> photos1 = manager.getPhotos();
    LinkedList<Photo> photos2 = new LinkedList<Photo>();

    if (!photos1.empty()) {
        photos1.findFirst();
        while (true) {
            photos2.insert(new Photo(photos1.retrieve().getPath(), photos1.retrieve().getTags()));
            if (photos1.last()) break;
            photos1.findNext();
        }
    }

    NbComps = 0;
    if (!condition.equals("")) {
        String[] requiredTags = condition.split("AND");
        if (!photos2.empty()) {
            photos2.findFirst();
            while (true) {
                Photo photo = photos2.retrieve();
                if (!allAvailable(photo.getTags(), requiredTags)) {
                    photos2.remove();
                    if (photos2.empty()) break;
                } else {
```

```
                if (photos2.last()) break;
                photos2.findNext();
            }
        }
    }
  }
  return photos2;
}
```

- allAvailable(LinkedList<String> AllTags, String[] arr)
  Verifies that a photo contains all required tags and counts comparisons:

```
private boolean allAvailable(LinkedList<String> AllTags, String[] arr) {
  boolean contin = true;
  if (AllTags.empty()) {
    contin = false;
  } else {
    for (int i = 0; i < arr.length && contin; i++) {
      boolean tagMatched = false;
      AllTags.findFirst();
      while (!AllTags.last()) {
        this.NbComps++;
        if (AllTags.retrieve().compareToIgnoreCase(arr[i]) == 0) {
          tagMatched = true;
          break;
        }
        if (AllTags.last()) break;
        AllTags.findNext();
      }
      if (!tagMatched) contin = false;
    }
  }
  return contin;
```

4.5 BST<T> Class

Data Representation

The Binary Search Tree (BST) is represented by:

- root: a reference to the root node.

- current: a reference to the currently accessed node.

Each BSTNode<T> contains:

- key: a String key (used for ordering).
- data: the associated value.
- left: pointer to the left child node.
- right: pointer to the right child node.

Operations Implementation

- empty()
  Checks if the tree is empty:

```
public boolean empty() {
   return root == null;
}
```

- findkey(String tkey)
  Searches for a key in the tree. Updates current accordingly:

```
public boolean findkey(String tkey) {
   // Traverses tree comparing tkey with current node's key
}
```

- insert(String key, T val)
  Inserts a new node while maintaining BST properties:

```
    public boolean insert(String key, T val) {
       // Uses findkey first to avoid duplicates.
    }
```

- retrieve()
  Returns the data stored in the current node:

```
public T retrieve() {
   return current.data;
}
```

- removeKey(String k)
  Removes a node by key, handling cases (0, 1, 2 children):

```
public boolean removeKey(String k) {
   // Iterative search and deletion by replacing with successor if needed.
}
```

- inOrder()
  Returns all keys in ascending order (inorder traversal):

```
public String inOrder() {
   inorder(root);
}
```

- update(String key, T data)
  Updates a node's data by removing and re-inserting:

```
public boolean update(String key, T data) {
   if (findkey(key)) removeKey(key);
   return insert(key, data);
}
```

4.6 InvIndexPhotoManager Class
Data Representation

The InvIndexPhotoManager uses a BST where:
- Each key is a tag (String).
- Each data is a LinkedList<Photo> storing photos associated with the tag.

Operations Implementation

- addPhoto(Photo p)
  Adds a photo to the inverted index. For each tag of the photo:
  - If the tag exists, add the photo to the existing LinkedList.
  - Otherwise, insert a new key-value pair.

- deletePhoto(String path)
  Deletes a photo from all tag entries:
  - Traverses all tags via inOrder.
  - Removes photo from the corresponding list.
  - Deletes the tag node if its list becomes empty.

- getPhotos()
  Returns the entire BST representing the inverted index.

4.7 InvAlbum Class

Data Representation

An InvAlbum contains:

- name: the album name.
- condition: a string of required tags.
- invmanager: a reference to the InvIndexPhotoManager.
- NbComps: counter for tag comparisons.

Operations Implementation

- getPhotos()

Retrieves photos based on the album's condition:

- If the condition is not empty, split tags and retrieves matching lists.

- If there is no condition, retrieves all photos in the index.

Two main helper functions are used:
- Fun1: Computes the intersection of two lists.
- Fun2: Computes the union of two lists.

Fun1(LinkedList<Photo> L1, LinkedList<Photo> L2)
Returns the intersection between L1 and L2.

Fun2(LinkedList<Photo> L1, LinkedList<Photo> L2)
Returns the union between L1 and L2.

getNbComps()
Returns the number of tag comparisons performed.

5. Performance analysis

The method Album.getPhotos() retrieves all photos that satisfy a given condition based on tags. The performance can be analyzed as follows:

Before inverted index:

First, the method copies all the photos from the PhotoManager to a temporary list.

- Copying n photos requires O (n) time, where n is the total number of photos.

Next, for each photo, the method checks whether it satisfies the album's condition:

- The condition contains k required tags (split by "AND").
- To check the required tag, the method needs to search through up to m tags of the photo.

Therefore, for one photo, checking all condition tags requires O (k * m) time.

Checking all n photos results in a total running time of O (n * k * m) so,

Big-O = **O (n * k * m)**

Where…

n = number of photos,

k = number of tags in the album's condition,

m = number of tags in one photo.

In the worst case, the method must scan through every tag of every photo for every condition tag, making the algorithm's running time proportional to n * k * m.

If k and m are considered small numbers compared to n, the complexity can be simplified to O(n).

After inverted index:

When an inverted index is used, the method getPhotos() becomes much faster.
Instead of checking each photo manually, the inverted index allows direct access to the list of photos associated with each required tag.

The way it works:

- For each required tag (k tags), the method retrieves the list of matching photos directly from the index. This retrieval is O (1) for each tag.
- After retrieving all k lists, the method computes the intersection of these lists to find photos that satisfy all required tags.

Assuming the average number of photos per tag is **s**, computing the intersection takes about **O (s * k)** operations.

The overall time complexity after using an inverted index is:

Big-O = **O(s * k)**

Where…

k = number of tags in the album's condition,

s = average number of photos having a required tag.

Since (s) is usually much smaller than (n), especially if the condition is specific, the method becomes much faster.

In conclusion, after using the inverted index, getPhotos() no longer depends on the total number of photos (n), but only on the size of the relevant tag lists (**s**) and the number of condition tags (k).

Example (for Clarity)

If we have:

- 10,000 photos in the system (**n = 10,000**),
- Album condition with 2 tags (**k = 2**),
- Each photo has 5 tags on average (**m = 5**).

Before inverted index:

- Need to check every photo manually.
- Total operations = 10,000 × 2 × 5 = 100,000 comparisons.
- Big-O = **O(n * k * m)**

After inverted index:

- Only about 300 photos match "animal" and 200 match "grass".
- Searching among these smaller lists.
- Total operations = 300 × 2 = 600 comparisons.
- Big-O = **O(s * k)**

Thus, the inverted index greatly reduces the number of operations.

6. Conclusion

We successfully designed and implemented a photo management application. The initial implementation was based on a simple linear structure using singly linked lists, where albums were dynamically generated by checking the presence of specified tags in each photo.

To optimize the retrieval process, we enhanced the system by constructing an Inverted Index using a Binary Search Tree (BST). This optimization significantly improved the efficiency of album creation by reducing the search complexity from linear time to logarithmic time in terms of tag lookup.

Limitations:

- The system currently supports only the "AND" logical operation between tags.
- Tags are case-insensitive but do not account for variations like synonyms or plurals.

Future Work:

- Extend the condition parser to support "OR" and "NOT" operations.
- Enhance the system with a graphical user interface (GUI) for better usability.
- Integrate advanced search features such as tag synonym recognition.

Working on this project helped us understand how data structures work in real life situations. It showed us how picking the right way to store and search for data can make a big difference in how fast and efficient a system runs.