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
خالد ناصر عزالدين	**444107021**	**Mostly InvIndexPhotoManager class**
البراء محمد عثمان	**444107015**	**Mostly Album, PhotoManger classes**
ياسر	**444100878**	**Mostly Photo, and data structure classes**

1. Introduction

photo management application

This application is used to organize photographs so they can be accessed easily, the basic idea that the user will provide tags to describe content of the photo

Ex :

 animal, bear, cab, grass, wind

In this report we will provide information about 4 topics:

1- **Specification :** we will provide in this topic every class in this program including the attributes and methods associate with each class and we will provide the specification of every data structure used in this program

2- **Design :** in this topic we will provide a detailed description of how the project was designed and we will not talk about any implementation (coding) here.

3- **Implementation :** in this topic we will provide a detailed description of the technical implementation (coding) only the critical parts.

4- performance : in this topic we will discuss and analyze the performance of the code (BIG-O Notation) specially the **getPhotos()** method**.**

In the end of this report our conclusion will be provided.

2. Specification

Classes:

1- Class Photo

Attributes:

private String path;

private LinkedList<String> tags;

Methods:

public Photo(String path, LinkedList tags); // Constructor

public String getPath(); // Return the full file name (the path) of the photo, A photo is uniquely identified by its path.

public LinkedList getTags(); // Return all tags associated with the photo

2- Class PhotoManager

Attributes:

private LinkedList<Photo> photos;

Methods:

public PhotoManager(); // Constructor

public LinkedList getPhotos(); // Return all managed photos

public void addPhoto(Photo p); // Add a photo

public void deletePhoto(String path); // Delete a photo

3- Class Album

Attributes:

private String name;

private String condition;

private PhotoManager manager;

private int nbComps;

Methods:

public Album(String name, String condition, PhotoManager manager); // Constructor

public String getName(); // Return the name of the album

public String getCondition(); // Return the condition associated with the album

public PhotoManager getManager(); // Return the manager

public LinkedList getPhotos(); // Return all photos that satisfy the album condition

public int getNbComps(); // Return the number of tag comparisons used to find all photos of the album

4- Class Node<T>

Attributes:

public T data;

public Node<T> next;

Methods:

public Node() //constructor

public Node(T data) // constructor

// Setters/Getters

public void setData(T data)

public T getData()

public void setNext(Node<T> next)

public Node<T> getNext()

5- Class LinkedList<T>

Attributes:

public Node<T> head;

public Node<T> current;

Methods:

public LinkedList() // constructor

public boolean empty() // Check if the linked list is empty

public boolean last() // Check if the current element is the last element

public boolean full() // check if the linked list is full(linked list is never full)

public void findFirst() // first element set as the current element

public void findNext() // the element next to the current element become the current element

public void update(T data) // update the data in the current element with

public T retrieve () // return the data in the current element

public void insert(T data) // insert a new element in the linked list after the current element

public void remove() // remove the current element in the linked list

6- Class BSTNode <T>

Attributes:

public int key;

public T data;

public BSTNode<T> left;

public BSTNode<T> right;

Methods:

public BSTNode(int k,T data) // constructor

public BSTNode(int k, T data, BSTNode<T> left, BSTNode<T> right) // constructor

7- Class BST<T>

Attributes:

BSTNode<T> root;

BSTNode<T> current;

Methods:

public BST() // constructor

public boolean empty() // check if the BST is empty

public T retrieve() // return the current element data

public boolean update(int key, T val) // current element data and key will replaced

public boolean insert(int k,T val) //insert a new element and place it in the tree based on the key

public void deleteSubtree() // delete the subtree that the current is the root of it

public boolean find(*Relative* rel) //

private BSTNode<T> findParent(BSTNode<T> p, BSTNode<T> t)

8- **Class InvIndexPhotoManager**

Attributes:

private BST<LinkedList<Photo>> invertedIndex;

private LinkedList<Photo> phoIn = new LinkedList<Photo>();

Methods:

public Photomanager(); // Constructor

public void addPhoto(Photo p); // Add a photo

public void deletePhoto(String path); // Delete a photo

public BST<LinkedList> getPhotos(); // Return the inverted index of all managed photos

9- **class Relative**

public enum *Relative* {***Root***, ***Parent***, ***LeftChild***, ***RightChild***};

10- **class Order**

public enum *Order* {***preOrder***, ***inOrder***, ***postOrder***};

classes 9 and 10 have what we call enum (short for "enumeration") is a special data type used to define a set of predefined constant values. It improves code readability and type safety by allowing a variable to take only one of the specified values.

Linked List ADTs specification:

Elements: The elements are of generic type (The elements are placed in nodes for linked list implementation).

Structure: the elements are linearly arranged. The first element is called head, there is an element called current.

Domain: the number of elements in the list is bounded therefore the domain is finite. Type name of elements in the domain: List

Operations: We assume all operations operate on a list L. 1.

1- **Method** findFirst ( )

requires: list L is not empty. **input:** none

results: first element set as the current element. **output:** none.

2- **Method** findNext ( )

requires: list L is not empty. Current is not last. **input:** none

results: element following the current element is made the current element. **output:** none.

3- **Method** retrieve (Type e)

requires: list L is not empty. **input:** none

results: current element is copied into e. **output:** element e.

4- **Method** update (Type e)

requires: list L is not empty. **input:** e.

results: the element e is copied into the current node. **output:** none.

5- **Method** insert (Type e)

requires: list L is not empty. **input:** e.

results: a new node containing element e is created and inserted after the current element in the list. The new element e is made the current element. If the list is empty e is also made the head element. **output:** none.

6- **Method** remove ( )

requires: list L is not empty. **input:** none

results: the current element is removed from the list. If the resulting list is empty current is set to NULL. If successor of the deleted element exists it is made the new current element otherwise first element is made the new current element. **output:** none.

7- **Method** full (boolean flag)

input: none

results: if the number of elements in L has reached the maximum number allowed then flag is set to true otherwise false. **output:** flag.

8- **Method** empty (boolean flag).

input: none

results: if the number of elements in L is zero, then flag is set to true otherwise false. **output:** flag.

8- **Method** last (boolean flag).

input: none. **requires:** L is not empty.

results: if the last element is the current element then flag is set to true otherwise false. **output:** flag.

BST ADTs specification:

Elements: the elements are nodes (BSTNode), each node contains the following data type: Type,Key and has LeftChild and RightChild references.

Structure: hierarchical structure; each node can have two children: left or right child; there is a root node and a current node. If N is any node in the tree, nodes in the left subtree < N and nodes in the right subtree > N.

Domain: the number of nodes in a BST is bounded; type/class name is BST

Operations:

1- **Method** FindKey (int tkey, boolean found)

requires: none. **input:** tkey.

results: If bst contains a node whose key value is tkey, then that node is made the current node and found is set to true; otherwise found is set to false and either the tree is empty or the current node is the node to which the node with key = tkey would be attached as a child if it were added to the BST.

output: found.

2- **Method** Insert (int key, Type e, boolean inserted)

requires: Full (bst) is false. **input:** key, e.

results: if bst does not contain k then inserted is set to true and node with k and e is inserted and made the current element; otherwise inserted is set to false and current value does not change.

output: inserted.

3- **Method** Remove_Key (int tkey, boolean removed)

input: tkey.

results: Node with key value tkey is removed from the bst and removed set to true. If BST is not empty then root is made the current. **output:** removed.

4- **Method** Update(int key, Type e, boolean updated)

requires: Empty(bst) is false. **input:** key, e.

results: current node's element is replaced with e. **output:** updated.

5- **Method** DeleteSub ()

requires: BST is not empty. **input:** none.

results: The subtree whose root node was the current node is deleted from the tree. If the resulting tree is not empty, then the root node is the current node. **output:** none.

6- **Method** Retrieve (Type e)

requires: BST is not empty. **input:** none.

results: element in the current node is copied into e. **output:** e.

7- **Method** Empty (boolean empty).

requires: none. **input:** none.

results: If Binary tree is empty then empty is true; otherwise, empty is false. **output:** empty.

8- **Method** Full (boolean full)

requires: none. **input:** none.

results: if the binary tree is full then full is true otherwise false. **output:** full.

9- **Method** Traverse (Order ord)

requires: : Binary Tree (BT) is not empty. **input:** ord.

results: Each element in the tree is processed exactly once by a user supplied procedure. The order in which nodes are processed depends on the value of ord (Order = {preorder, postorder, inorder})

preorder: each node processed before any node in either of its subtrees.

inorder: each node is processed after all its nodes in its left subtree and before any node in its right subtree.

postorder: each node is processed after all its nodes in both of its subtrees.

output: none.

3. Design

This project has 2 main things the first one is that we will have an album and number of photos that have a name(path) and number of tags in the system, and we will not indicate whether these photos are met the condition of the album or not unless we ask for the photos in the album, and if any photo met the condition we will add it in the album and after we add all photos that met the condition of the album we give the photos that added to the album.

For example:

Photo1: Hedgehog.jpg tags: grass

Photo1: bear.jpg tags: grass, animal, bear

Album1: alb1 condition: animal

The photos that will be in alb1 is: bear.jpg only.

Second thing is the inverted index which is different from what an album is doing, in the inverted index we will make every tag point on number of photos that this tag is associated with them.

for example:

animal → hedgehog.jpg, bear.jpg, fox.jpg, panda.jpg, wolf.jpg, racoon.jpg

apple → hedgehog.jpg

bear → bear.jpg, panda.jpg

4. Implementation

[Write a detailed description of the technical implementation details. The description should cover only the critical implementation parts that need to be clarified. Use code snippets if needed]

We use for class photo 2 attributes one is the path and the other is a linked list of tags and this allow us to implement photoManager class which we use it in album class, but to implement the class InvIndexPhotoManager we implement first the BST class which give us a search function that is O(log n) in avg case.

In InvIndexPhotoManager we use BST<LinkedList<Photo>> index attribute to store all photos based on the key which is a tag in every node so, when we access every node we can see the photos belongs to a certain tag for ex:

```
animal-> hedgehog.jpg, bear.jpg
hedgehog-> hedgehog.jpg
apple-> hedgehog.jpg
grass-> hedgehog.jpg, bear.jpg
```

In InvIndexPhotoManager class

Add photo:

```
/* Complexity is O(n) where n is the number of tags in the photo.
public void addPhoto(Photo p) {
    LinkedList<String> tags = p.getTags();
    if(p == null)
        return;
    if(tags.empty())
        return;
    if(findPhoto(p))
        return;
    String tag;
    LinkedList<Photo> ph ;
    LinkedList<String> t;
    Photo p1;
    tags.findFirst();
    while(!(tags.last())) {
        tag = tags.retrieve();
        if(index.empty() ||!index.findkey(tag)) {
            ph = new LinkedList<Photo>();
            t = new LinkedList<String>();
            t.insert(tag);
            p1 = new Photo(p.getPath(), t);
            ph.insert(p1);
            tags.findNext();
            index.insert(tag, ph);
        }
        else {

                t = new LinkedList<String>();
                t.insert(tag);
                p1 = new Photo(p.getPath(), t);
                tags.findNext();
                index.retrieve().insert(p1);
```

```
                    t = new LinkedList<String>();
                    t.insert(tag);
                    p1 = new Photo(p.getPath(), t);
                    tags.findNext();
                    index.retrieve().insert(p1);

            }
        }
        tag = tags.retrieve();
        if(index.empty()||!index.findkey(tag)) {
            ph = new LinkedList<Photo>();
            t = new LinkedList<String>();
            t.insert(tag);
            p1 = new Photo(p.getPath(), t);
            ph.insert(p1);
            tags.findNext();
            index.insert(tag, ph);
        }

            else {
                t = new LinkedList<String>();
                t.insert(tag);
                p1 = new Photo(p.getPath(), t);
                tags.findNext();
                index.retrieve().insert(p1);
            }

    phoIn.insert(p);

    }
```

Get photos:

```
public BST<LinkedList<Photo>> getPhotos(){
    return index;
}
```

Delete photos:

```
public void deletePhoto(String path) {
    if(!findPhoto(path)) {
        System.out.println("No photos with this path is here");
        return;
    }
    delete(path);
    phoIn.remove();


}
private void delete(String path) {
   LinkedList<String> t = phoIn.retrieve().getTags();
   t.findFirst();
   String s;
   while(true) {
       s = t.retrieve();
       if(index.findkey(s)) {
           index.retrieve().findFirst();
           while(true) {
           if(index.retrieve().retrieve().getPath().equalsIgnoreCase(path)) {
               index.retrieve().remove();
               break;
           }
           index.retrieve().findNext();
           }
           if(index.retrieve().empty())
               index.remove_key(s);
           }
       if(t.last())
           break;
       t.findNext();
   }
}
```

In album class

Get photos:

```java
public LinkedList<Photo> getPhotos() {
    LinkedList<Photo> result = new LinkedList<>();
    LinkedList<Photo> allPhotos = manager.getPhotos();
    nbComps = 0;

    if (condition == null || condition.equals("") || condition.trim().isEmpty()) {
        return allPhotos; // If the condition is empty, return all photos
    }

    String[] requiredTags = condition.split("AND");
    for (int i = 0; i < requiredTags.length; i++) {
        requiredTags[i] = requiredTags[i].trim(); // Remove leading and trailing spaces
    }

    allPhotos.findFirst();
    while(true) {
        Photo photo = allPhotos.retrieve();
        LinkedList<String> tags = photo.getTags();

        if (tags.empty())
            continue; // Skip photos with no tags
        boolean matches = true;

        tags.findFirst();
        for (int j = 0; j < requiredTags.length; j++) {
            boolean tagFound = false;
            while(true) {

                if (tags.retrieve().equalsIgnoreCase(requiredTags[j])) {
                    tagFound = true;
                    nbComps++;
                    break;
                }
                if (!tags.last()) {
                    tags.findNext();
                    nbComps++;
                }
```

```
                    nbComps++;
            }
                else
                    break;

            }
            if (tagFound) {
                if(result.empty())
                    result.insert(photo);
                while(true) {
                    if(photo.getPath().equalsIgnoreCase(result.retrieve().getPath())) {
                        matches = false;
                        break;
                    }
                    if (!result.last())
                        result.findNext();

                    else
                        break;

                }
                if(matches)
                result.insert(photo);
            }
        }
        if (!allPhotos.last())
            allPhotos.findNext();
        else
            break;
    }
    return result;
}
```

5. Performance analysis

we have in album.getphotos a big o of (n square) but after invertedIndex implemented the big o if we see it from extrnal it's big o of 1 but from internal the big o is also n square but we can see that that add in avg case is log n and delete in avg case is n log n which we can see an improvement here.

6. Conclusion

We can see that in first half of the project easier to implement since it's linked list but the cost is a bigger big o than the second half of the project which we used BST that allow us to implement a search function that it's avg case is big o of log n.

Github link: https://github.com/Baraa-Othman/CSC212Project/tree/main