**Department of Computer science and Engineering**

**CS 204:Design and Analysis of Algorithm**

**Project Title:OBE Implementation**

***Team Deatail****s:*

**Team Name :** Sunrisers

**Team project:** Program School

**Team members:**

K.Anishitha - AP23110011039

V.Hanshika - AP23110011031

K.Hanisha - AP23110011042

Syed Abdul Basith - AP23110011061

P.Hemanth - AP23110011057

***SUBMITTED TO***

**Designed by**

**GAVASKAR S,**

**Assistant Professor(Ad),**

**Department of CSE,**

**SRM University - AP.**

**INDEX**

**Introduction**

**Modules in the project:**

**Architecture Diagram**

Instructions:

**Module Description**

Programming Details naming conventions to be used:

Field/table details : Program Schools

Algorithm Details:

(i)Sorting

(ii)Searching

(ii) Storing the details in a text file 6

**Sample coding Template**

OBE\_MAIN\_SUNRISERS.c

Instruction to use Sample Code:

**ChatGPT Usage**

Program Generated by ChatGPT

Instruction to use ChatGPT or other LLM Models

**Introduction**

This C++ program is a student management system that uses a linked list to store and manage student records, including IDs, names, subjects, and marks. It allows users to add, update, remove students, and generate detailed reports for individual students or the entire list. The system includes features for searching student records by ID and sorting students based on their IDs or other criteria. Additionally, it supports saving and retrieving data from files, ensuring persistence across sessions. Users can set and display school details, such as the school name and location. The program operates through an interactive menu, providing a user-friendly way to manage student data, update marks, and generate reports, while also offering efficient searching and sorting capabilities for quick access to student information.

**Project Module:**

**Program Schools**

The primary module in this project is Program Management, focused on handling records for different programs.

This module allows:

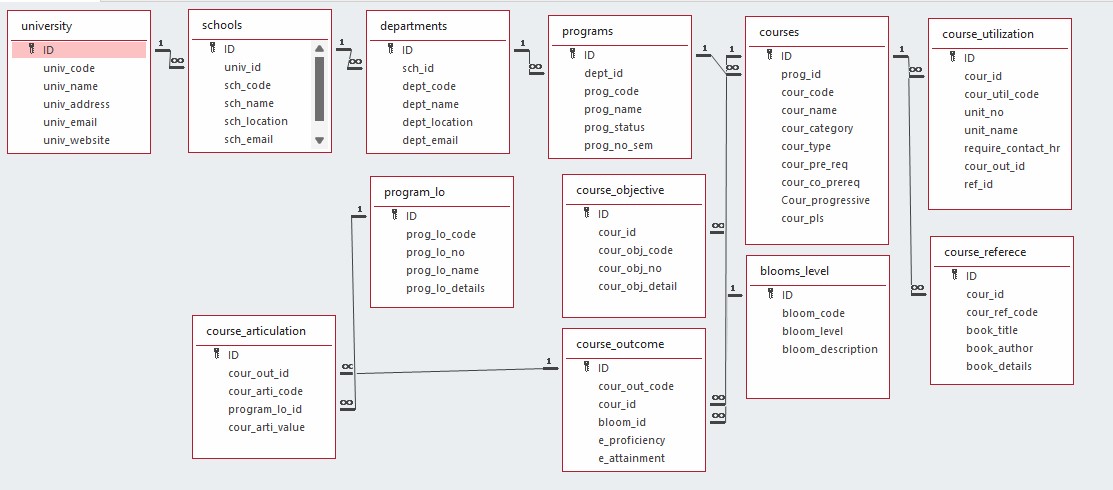
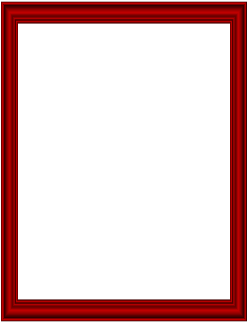
**ID-based identification**: Each program has a unique integer ID.

**Attribute-based searching and sorting**: Programs can be located and organized by their code, name for easier access and management.

**CRUD Operations**: Users can create, retrieve, update, and delete program records efficiently.

The Program Management module leverages Bubble Sort for sorting and Linear Search for basic searching, providing a straightforward approach to organizing and locating records.

**Architecture Diagram**



# 

# **Module Description**

**Module Name:** *Program Management*

**Module Description:**

The Program Management module facilitates CRUD operations on program records. Users can:

**Add a new program record.**

**View (retrieve) existing records.**

**Update specific program details.**

**Delete program records.**

Data is stored in a file (programs\_setting.txt), and each change (create, update, delete) is reflected in this file. Sorting and searching capabilities are also provided, allowing users to organize and locate program records based on attributes like program code and name.

Programming Details naming conventions to be used:

* **File name:** ● **Function/method name**

○ **Create:** The\_Sunrisers\_create\_program

○ **Update:** The\_Sunrisers \_update\_program

○ **Retrieve:** The\_Sunrisers \_retrieve\_programs

○ **Delete:** The\_Sunrisers \_delete\_program

○**Sorting:** The\_Sunrisers \_sort\_by\_field

○**Searching:** The\_Sunrisers \_search\_by\_field

○ **Comparison(both Searching & Sorting)**:

■ For Searching - The\_Sunrisers \_compare\_searching\_algorithms

■ For Sorting - The\_Sunrisers \_compare\_sorting\_algorithms

○ **Time Complexity(both searching and Sorting):**

■ For Searching/Sorting- The\_Sunrisers \_display\_time\_complexity

○ **Algorithm Details(pseudocode or steps)(both searching and Sorting):**

■ For Searching: The\_Sunrisers \_display\_pseudocode

●**File name(for storing the details):**

○ File name to be used is:- Programs\_setting .txt

Field/table details: For Programs

|  |  |
| --- | --- |
| **Field Name** | **Data type** |
| id | Char |
| School\_code | char |
| School\_name | char |
| School\_location | char |

Algorithm Details:

**1. Setting School Details**

* **Input**: name and location of the school.
* **Process**:
  + Assign the provided name and location values to the schoolName and schoolLocation member variables.
* **Output**: Confirmation message displaying that the school details were successfully updated.

**2. Adding a Student**

* **Input**: Student object containing id, name, and a map of subjects with marks.
* **Process**:
  + Create a new Node with the Student data.
  + Set the next pointer of this new node to point to the current head of the linked list.
  + Update head to point to this new node.
* **Output**: Confirmation message that the student was successfully added.

**3. Removing a Student**

* **Input**: studentId (the ID of the student to remove).
* **Process**:
  + Traverse the linked list from the head:
    - If a node with the matching studentId is found:
      * Update the next pointer of the previous node to skip the node containing the studentId.
      * If the node to be removed is the head, simply update head to the next node.
      * Free the memory of the node.
      * Return a success message.
    - If the end of the list is reached without finding the studentId, return a "student not found" message.
* **Output**: Success message if removed or an error message if not found.

**4. Printing a Single Student Report**

* **Input**: studentId (the ID of the student whose report is requested).
* **Process**:
  + Traverse the linked list to locate the node with the matching studentId.
  + If found, print the student’s ID, name, and subject marks.
  + If not found, print an error message.
* **Output**: Student report or an error message.

**5. Printing All Student Reports**

* **Input**: None.
* **Process**:
  + Traverse the entire linked list from head.
  + For each node, print the student’s ID, name, and their subject marks.
  + If the list is empty, print a message indicating no students to display.
* **Output**: Reports for all students or a message indicating an empty list.

**6. Saving to a File**

* **Input**: filename (name of the file to save data to).
* **Process**:
  + Open the file in binary write mode.
  + Write the length of schoolName and schoolLocation and their contents to the file.
  + Traverse each node in the linked list:
    - Write the student's id and name.
    - For each subject, write the subject name and the mark.
  + Close the file.
* **Output**: Confirmation message indicating data was saved.

**7. Retrieving from a File**

* **Input**: filename (name of the file to load data from).
* **Process**:
  + Open the file in binary read mode.
  + Read the length and content of schoolName and schoolLocation.
  + Clear the existing linked list.
  + Read each student’s data, create a new Node for each, and add it to the linked list.
  + Close the file.
* **Output**: Confirmation message indicating data was loaded successfully.

**8. Updating Marks for a Specific Subject**

* **Input**: studentId (ID of the student), subject name, and newMark.
* **Process**:
  + Locate the node with the specified studentId.
  + If found, update the subject's mark with newMark in the subjects map.
  + If not found, print an error message.
* **Output**: Success message if updated or an error message if student not found.

**9. Finding a Student by ID**

* **Input**: studentId (ID of the student to locate).
* **Process**:
  + Traverse the linked list.
  + Return a pointer to the Student object if a node with the matching studentId is found.
  + If not found, return nullptr.
* **Output**: Pointer to Student object or nullptr if not found.

**10. Destructor for LinkedList**

* **Process**:
  + Traverse each node in the linked list starting from head.
  + Delete each node to free allocated memory.
* **Output**: No output; ensures memory cleanup upon program exit.

**Source Code**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

// School structure

typedef struct {

int sch\_id;

char sch\_code[10];

char sch\_name[50];

char sch\_location[100];

char sch\_email[50];

} School;

// Global array to store school data and a counter

School schools[MAX];

int school\_count = 0;

// File name for storing the details

const char\* FILE\_NAME = "school\_setting.txt";

// Function declarations

void sunrisers\_create();

void sunrisers\_update();

void sunrisers\_retrieve();

void sunrisers\_delete();

void sunrisers\_storing();

void sunrisers\_sortbycode();

void sunrisers\_searchbycode();

void sunrisers\_linearsearchbyname();

void sunrisers\_complexity\_searching();

void sunrisers\_complexity\_sorting();

void merge(School arr[], int l, int m, int r);

void mergeSort(School arr[], int l, int r);

// Function to load data from the file into the schools array

void load\_from\_file() {

FILE \*file = fopen(FILE\_NAME, "r");

if (file == NULL) {

return; // No file exists yet

}

school\_count = 0;

while (fscanf(file, "%d %s %s %s %s\n", &schools[school\_count].sch\_id,

schools[school\_count].sch\_code,

schools[school\_count].sch\_name,

schools[school\_count].sch\_location,

schools[school\_count].sch\_email) != EOF) {

school\_count++;

}

fclose(file);

}

// Function to save data to the file

void sunrisers\_storing() {

FILE \*file = fopen(FILE\_NAME, "w");

if (file == NULL) {

printf("Error opening file!\n");

return;

}

for (int i = 0; i < school\_count; i++) {

fprintf(file, "%d %s %s %s %s\n", schools[i].sch\_id,

schools[i].sch\_code, schools[i].sch\_name,

schools[i].sch\_location, schools[i].sch\_email);

}

fclose(file);

}

// Function to create a school record

void sunrisers\_create() {

if (school\_count >= MAX) {

printf("School list is full!\n");

return;

}

School s;

printf("Enter School ID: ");

scanf("%d", &s.sch\_id);

printf("Enter School Code: ");

scanf("%s", s.sch\_code);

printf("Enter School Name: ");

scanf("%s", s.sch\_name);

printf("Enter School Location: ");

scanf("%s", s.sch\_location);

printf("Enter School Email: ");

scanf("%s", s.sch\_email);

schools[school\_count++] = s;

sunrisers\_storing();

printf("School created successfully!\n");

}

// Function to update a school record

void sunrisers\_update() {

int id;

printf("Enter School ID to update: ");

scanf("%d", &id);

for (int i = 0; i < school\_count; i++) {

if (schools[i].sch\_id == id) {

printf("Enter new School Code: ");

scanf("%s", schools[i].sch\_code);

printf("Enter new School Name: ");

scanf("%s", schools[i].sch\_name);

printf("Enter new School Location: ");

scanf("%s", schools[i].sch\_location);

printf("Enter new School Email: ");

scanf("%s", schools[i].sch\_email);

sunrisers\_storing();

printf("School updated successfully!\n");

return;

}

}

printf("School with ID %d not found.\n", id);

}

// Function to retrieve all school records

void sunrisers\_retrieve() {

printf("\nList of Schools:\n");

for (int i = 0; i < school\_count; i++) {

printf("ID: %d\nCode: %s\nName: %s\nLocation: %s\nEmail: %s\n\n",

schools[i].sch\_id, schools[i].sch\_code,

schools[i].sch\_name, schools[i].sch\_location,

schools[i].sch\_email);

}

}

// Function to delete a school record

void sunrisers\_delete() {

int id;

printf("Enter School ID to delete: ");

scanf("%d", &id);

for (int i = 0; i < school\_count; i++) {

if (schools[i].sch\_id == id) {

for (int j = i; j < school\_count - 1; j++) {

schools[j] = schools[j + 1];

}

school\_count--;

sunrisers\_storing();

printf("School deleted successfully!\n");

return;

}

}

printf("School with ID %d not found.\n", id);

}

// Function to search school by code

void sunrisers\_searchbycode() {

char code[10];

printf("Enter School Code to search: ");

scanf("%s", code);

for (int i = 0; i < school\_count; i++) {

if (strcmp(schools[i].sch\_code, code) == 0) {

printf("ID: %d\nCode: %s\nName: %s\nLocation: %s\nEmail: %s\n",

schools[i].sch\_id, schools[i].sch\_code,

schools[i].sch\_name, schools[i].sch\_location,

schools[i].sch\_email);

return;

}

}

printf("School with code %s not found.\n", code);

}

// Linear search by school name

void sunrisers\_linearsearchbyname() {

char name[50];

printf("Enter School Name to search: ");

scanf("%s", name);

for (int i = 0; i < school\_count; i++) {

if (strcmp(schools[i].sch\_name, name) == 0) {

printf("ID: %d\nCode: %s\nName: %s\nLocation: %s\nEmail: %s\n",

schools[i].sch\_id, schools[i].sch\_code,

schools[i].sch\_name, schools[i].sch\_location,

schools[i].sch\_email);

return;

}

}

printf("School with name %s not found.\n", name);

}

// Function to display time complexity of searching algorithms

void sunrisers\_complexity\_searching() {

printf("Linear Search: time complexity is O(n).\n");

printf("Binary Search (for sorted data): time complexity is O(log n).\n");

}

// Function to display time complexity of sorting algorithms

void sunrisers\_complexity\_sorting() {

printf("Quick Sort: time complexity is O(n^2) for the worst case and O(n log n) for the average case.\n");

printf("Merge Sort: time complexity is O(n log n).\n");

}

// Merge function for merge sort

void merge(School arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

School L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

if (strcmp(L[i].sch\_code, R[j].sch\_code) <= 0) {

arr[k++] = L[i++];

} else {

arr[k++] = R[j++];

}

}

while (i < n1) {

arr[k++] = L[i++];

}

while (j < n2) {

arr[k++] = R[j++];

}

}

// Merge sort function

void mergeSort(School arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

// Sort by school code

void sunrisers\_sortbycode() {

mergeSort(schools, 0, school\_count - 1);

printf("Schools sorted by code!\n");

sunrisers\_retrieve();

}

int main() {

load\_from\_file();

int choice;

while (1) {

printf("\n1. Create School\n2. Update School\n3. Retrieve Schools\n4. Delete School\n5. Search by Code\n6. Sort by Code\n7. Search by Name\n8. Show Search Complexity\n9. Show Sort Complexity\n10. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

sunrisers\_create();

break;

case 2:

sunrisers\_update();

break;

case 3:

sunrisers\_retrieve();

break;

case 4:

sunrisers\_delete();

break;

case 5:

sunrisers\_searchbycode();

break;

case 6:

sunrisers\_sortbycode();

break;

case 7:

sunrisers\_linearsearchbyname();

break;

case 8:

sunrisers\_complexity\_searching();

break;

case 9:

sunrisers\_complexity\_sorting();

break;

case 10:

exit(0);

default:

printf("Invalid choice!\n");

}

}

return 0;

}

# **Comparison of Sorting Algorithms**

1. **QuickSort**

QuickSort(arr[], low, high)

if low < high

pivot = Partition(arr[], low, high)

QuickSort(arr[], low, pivot - 1)

QuickSort(arr[], pivot + 1, high)

Partition(arr), low, high)

pivot = arr[high]

i = low-1

for j = low to high - 1 if arr[j] is less than pivot based on the sorting criteria

swap arr[i] with arr[j]

i++

swap arr[i + 1] with arr[high]

return i + 1

2**. Merge Sort**

void sunrisers\_schools\_merge(int left, int mid, int right) { int nl mid-left + 1, n2 = right - mid; School "L = malloc(n1 sizeof(School)), R = malloc(n2 sizeof(School));

for (int i = 0 i < n; i++) L[i] = schools[left + i]; for (int i = 0 i < n2; i++) R[i] = schools[mid + 1 + i];

int i = 0 j = 0 k = 1 left; while (i < n1 && j < n2) schools[k++] = strcmp(L[i].sch\_name, R[j].sch\_name) <= 0 ? L[i++]: R[j++];

while (i < n1) schools[k++] = L[i++]; while (j < n2) schools [k++] = R[j++];

free(L); free(R); }

void sunrisers\_schools\_merge\_sort(int left, int right) { if (left right) { int mid left + (right-left)/2; sunrisers\_schools\_merge\_sort(left, mid); sunrisers\_schools\_merge\_sort(mid + 1, right); sunrisers\_schools\_merge(left, mid, right);

}

}

**Time Complexity:**

**Quick Sort**

**Best Case: O(nlogn) - Pivot divides array into nearly equal halves.**

**Average Case: O(nlogn) - Pivot generally gives balanced splits.**

**Worst Case: O(n^2) - Pivot repeatedly gives unbalanced splits (e.g., sorted array)**

**Merge Sort**

**Best, Average, Worst Case: O(nlogn) - Always splits array into two equal halves and merges, so performance remains stable regardless of data order.**

**Why Merge Sort is Consistent:**

**Since it does not depend on pivot selection, Merge Sort's performance remains unaffected by the initial order of elements. It always divides evenly, ensuring balanced recursion in all cases.**

# **Comparison of Searching Algorithms**

**1. Linear Search (Primary Algorithm**)

**Approach: Sequentially checks each element in the list until the target is found**

**Time Complexity:**

**Best Case: O(1)O(1) - Target is the first element.**

**Average/Worst Case: O(n)O(n) - Target is in the middle or not present.**

**Requirements: Works on both sorted and unsorted lists.**

**Use Case: Good for small or unsorted datasets.**

**2. Binary Search (Comparison Algorithm)**

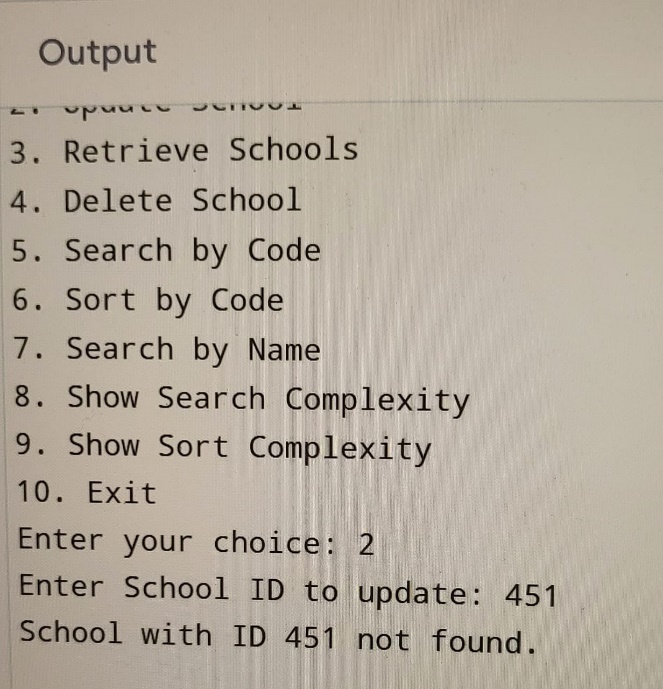
**Approach: Repeatedly divides a sorted list in half, checking if the target is in the left or right half.**

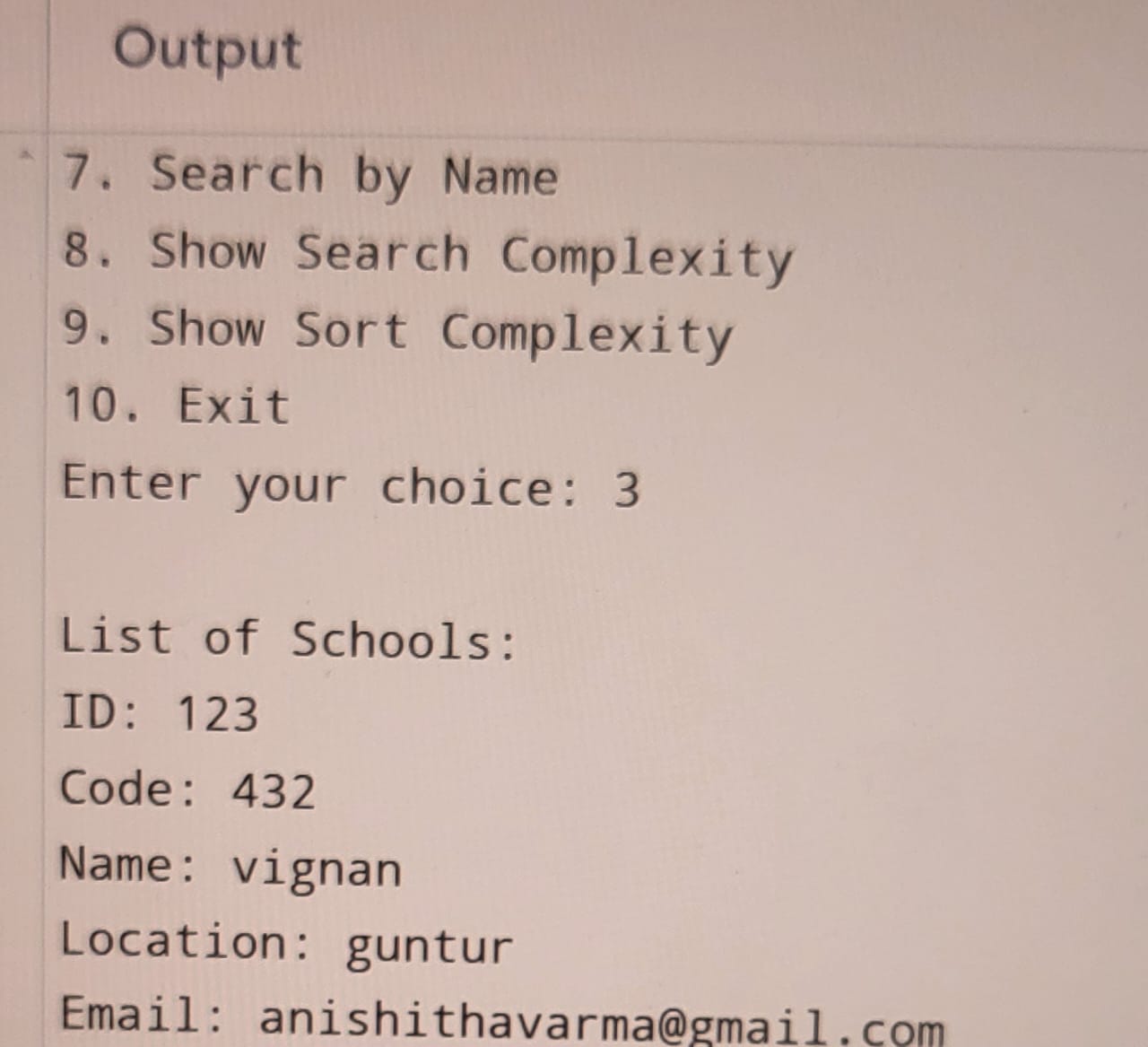
**Time Complexity: Best Case: O(1)O(1) - Target is the middle element. Average/Worst Case: O(logn) O(logn) Reduces search space by half each time.**

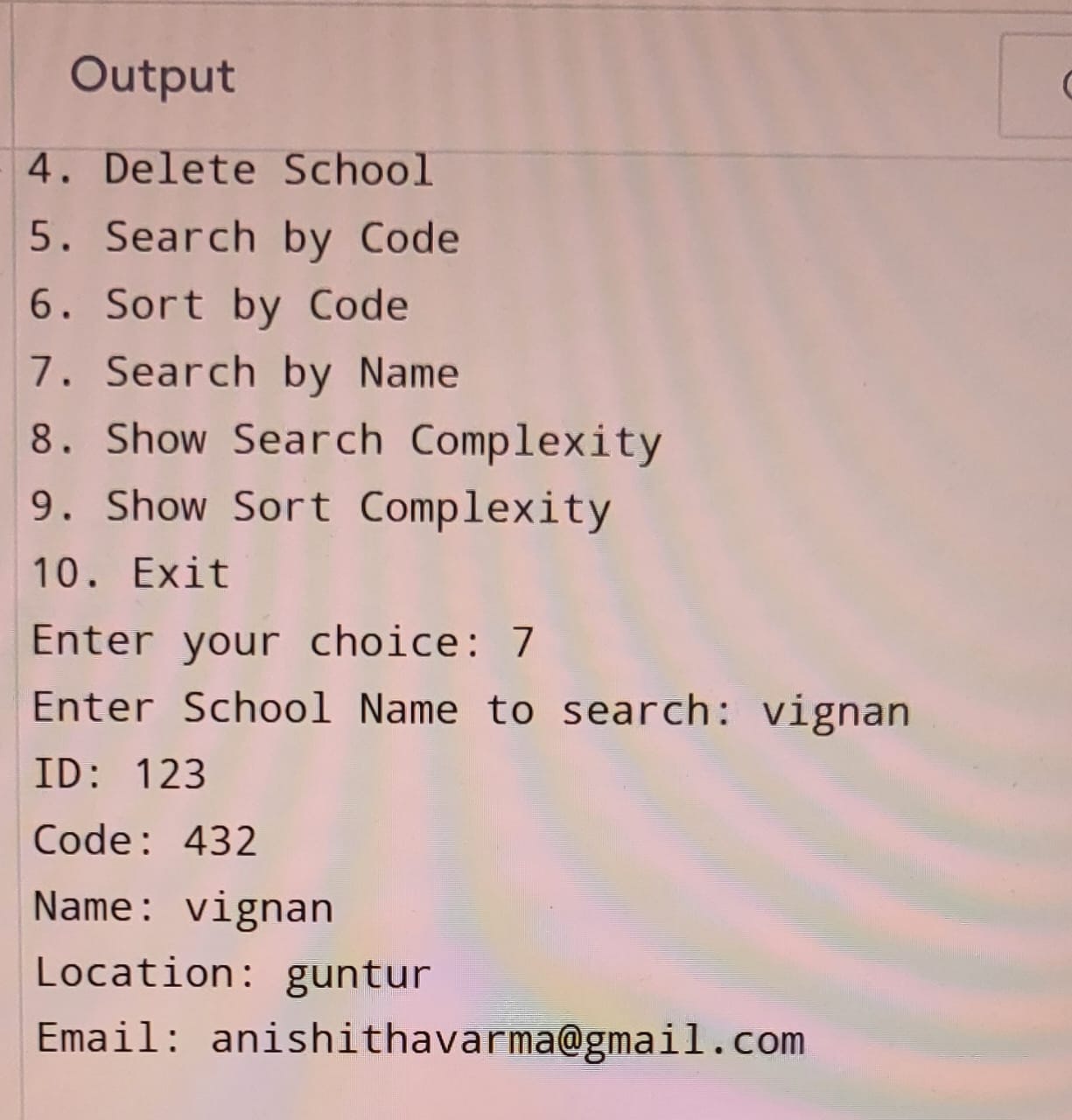
**Requirements: Requires a sorted list**

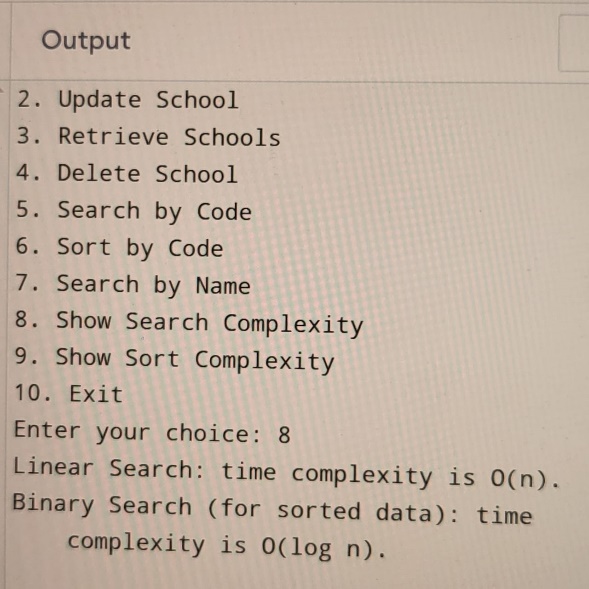
# **Output Screen Shots**

# 









# **CONCLUSION**

This project creates a simple and effective system to manage bloom level data, allowing users to store, view, update, and delete information as needed. To keep data organized and easy to find, the program includes sorting and searching methods. Merge Sort is used for its steady and reliable performance, while Quick Sort provides faster sorting for unsorted data. For searching, Binary Search is used for quick lookups on sorted data, and Linear Search allows for flexible searches on unsorted data.

Overall, the project efficiently handles data operations and keeps information saved in a file, making sure that changes are kept even after the program closes. This project demonstrates basic principles of data management, showing how to choose and use different algorithms based on specific needs to create a reliable system for managing bloom level data.

THANK YOU!