OBE IMPLEMENTATION: SCHOOLS

by

P. Srinidhi [AP23110011027]
I. Vinuthna [AP23110011044]
G. Venkat [AP23110011064]
S. Vamsi [AP23110011076]
D. Harsha Vardhan [AP23110011026]
P.V.N.K. Sreehitha [AP23110011037]

A report for the CS204:Design and Analysis of Algorithm project

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SRM UNIVERSITY AP::AMARAVATI

INDEX

Introduction

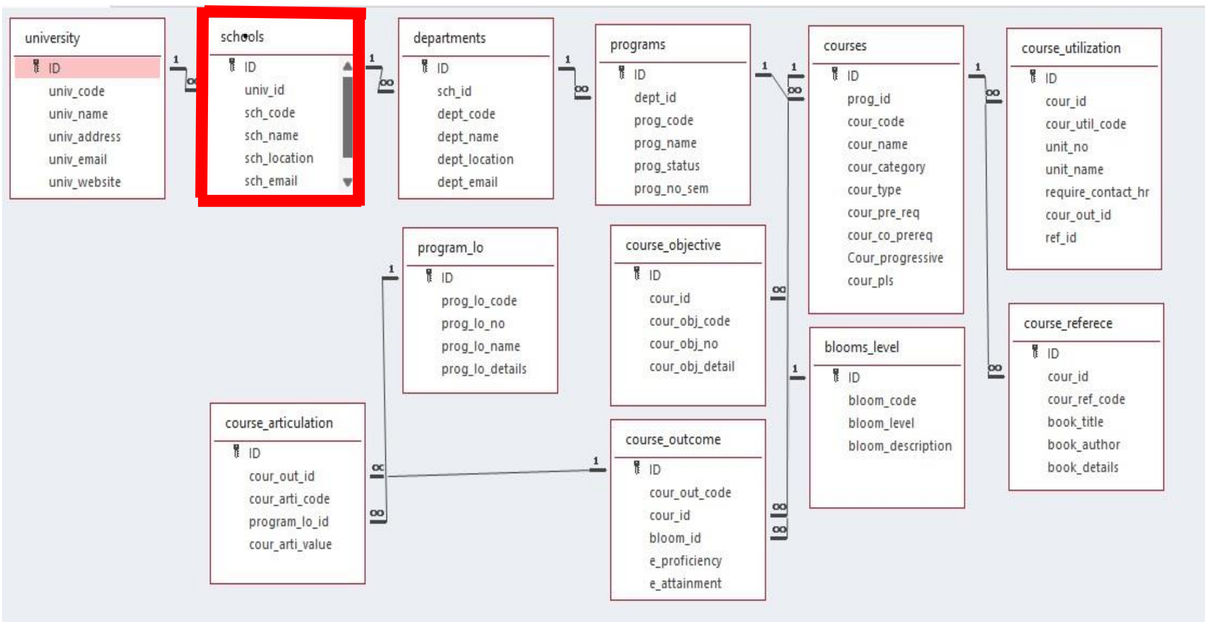
This C++ program is a student management system that uses a linked list to store and manage student records, including IDs, names, subjects, and marks. It allows users to add, update, remove students, and generate detailed reports for individual students or the entire list. The system includes features 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 Modules:

Various Modules available in the project are

1.Blooms Level setting
2.Program Level Objective Setting
3.University
4.Schools
5.Department
6.Programs
7.Courses
8.Course objective setting
9.Course Outcome Setting
10.Course Articulation matrix Setting
11. course Utilization Setting
12. Course Reference Setting

Architecture Diagram



Module Description

Module Name: Schools

Description:

This module is used to create,Update,Retrieve,Delete(hereafter known as CURD) details of the module and storing the details in the text file.you have to provide option for searching and sorting of fields mentioned below according to algorithms given for you

Programming Details naming conventions to be used:

- **File name:**The_Dreamcoders_Schools ●
  Function/method name
    - **Create: The_Dreamcoders_create_schools**
    - **Update: The_Dreamcoders_update_schools**
    - **Retrieve: The_Dreamcoders_retrieve_schools**
    - **Delete: The_Dreamcoders_retrieve_schools**
    - **Sorting: The_Dreamcoders_sort_by_bubblesort**
    - **Searching: The_Dreamcoders_search_by_binarysearch**
    - **Comparison(both searching and Sorting)**:
        - For Searching-The_Dreamcoders_Schools_Compare_Search_Binary Search
        - For Sorting- The_Dreamcoders_Schools_Compare_Sorting_Bubble Sort

        - **Time Complexity(both searching and Sorting):**
            - For Searching- The_Dreamcoders_Schools_O(logn)_Search_Binary Search
            - For Sorting- The_Dreamcoders_Schools_O(n^2)_Sorting_Bubble Sort
    - **Algorithm Details(pseudocode or steps)(both searching and Sorting):**
        - For Searching-The_Dreamcoders_Schools_Compare_Search_Binary Search
        - For Sorting- The_Dreamcoders_Schools_Compare_Sorting_Bubble Sort
- **File name(for storing the details)**
    - File name to be used is:-schools_setting .txt

Field/table details:

Field Name	Data type
id	Integer
school_code	String
school_name	String
school_address	String
school_email	String
school_website	String

Algorithm Details:

(i)Sorting

- Sorting used is BUBBLE SORT

Algorithm Steps:

1. **Input:** An array arr[] of n elements.

2. **Outer Loop:** Run a loop for i from 0 to n-1.

Each iteration of this loop represents one pass through the array.

3. **Inner Loop:** Run a loop for j from 0 to n-i-2.

Compare adjacent elements arr[j] and arr[j+1]. If arr[j] > arr[j+1], swap them.

(ii)Searching

- Searching used is BINARY SEARCH

Here are the steps for the binary search algorithm:

1. Initialize:

Set two pointers: low to the first index (0) and high to the last index (n-1) of the sorted array.

2. Repeat until low ≤ high:

Calculate the middle index: mid = low + (high - low) / 2.

Compare the middle element (arr[mid]) with the target value (key).

3. Decision:

If arr[mid] == key, return mid (index of the target element).

If arr[mid] > key, update high = mid - 1 (search the left half).

If arr[mid] < key, update low = mid + 1 (search the right half).

4. Terminate:

If low > high, the target is not in the array. Return an indicator (e.g., -1 for not found).

Complexity:

Time Complexity: O(log n)

Space Complexity: O(1) (iterative approach) or O(log n) (recursive approach due to stack space).

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define max 100

struct School {
    int school_Id;
    char school_code[10];
    char school_name[20];
    char school_address[30];
    char school_email[15];
    char school_website[10];
} typedef sch;

sch schools[max];
int school_count = 0;
const char *filename = "school_data.txt"; // Text file to store data

// Function to save data to a text file
void save_to_file() {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        printf("Error opening file for writing.\n");
        return;
    }

    fprintf(file, "%d\n", school_count);
    for (int i = 0; i < school_count; i++) {
        fprintf(file, "%d %s %s %s %s %s\n", schools[i].school_Id,
                schools[i].school_code, schools[i].school_name,
                schools[i].school_address, schools[i].school_email,
                schools[i].school_website);
    }

    fclose(file);
    printf("Data saved to file successfully.\n");
}

// Function to load data from a text file
void dream_coders_load_from_file() {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("No previous data found. Starting fresh.\n");
```

```
        return;
    }

    fscanf(file, "%d", &school_count);
    for (int i = 0; i < school_count; i++) {
        fscanf(file, "%d %s %s %s %s %s", &schools[i].school_Id,
               schools[i].school_code, schools[i].school_name,
               schools[i].school_address, schools[i].school_email,
               schools[i].school_website);
    }

    fclose(file);
    printf("Data loaded from file successfully.\n");
}

void getschooldetails(sch *s) {
    printf("Enter School ID: ");
    scanf("%d", &s->school_Id);
    printf("Enter School Code: ");
    scanf("%s", s->school_code);
    printf("Enter School Name: ");
    scanf("%s", s->school_name);
    printf("Enter School Address: ");
    scanf("%s", s->school_address);
    printf("Enter School Email: ");
    scanf("%s", s->school_email);
    printf("Enter School Website: ");
    scanf("%s", s->school_website);
}

void showschooldetails(sch *s) {
    printf("%d %s %s %s %s %s\n", s->school_Id, s->school_code, s->school_name,
s->school_address, s->school_email, s->school_website);
}

// Bubble Sort for Sorting the schools by school_Id
void dream_coders_bubble_sort() {
    int i, j;
    sch temp;
    for (i = 0; i < school_count - 1; i++) {
        for (j = 0; j < school_count - 1 - i; j++) {
            if (schools[j].school_Id > schools[j + 1].school_Id) {
                temp = schools[j];
                schools[j] = schools[j + 1];
                schools[j + 1] = temp;
            }
```

```
        }
    }
}

// Quick Sort for Sorting the schools by school_Id
void dream_coders_quick_sort(int low, int high) {
    if (low < high) {
        int pivot = schools[high].school_Id;
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (schools[j].school_Id < pivot) {
                i++;
                sch temp = schools[i];
                schools[i] = schools[j];
                schools[j] = temp;
            }
        }
        sch temp = schools[i + 1];
        schools[i + 1] = schools[high];
        schools[high] = temp;
        int pi = i + 1;
        dream_coders_quick_sort(low, pi - 1);
        dream_coders_quick_sort(pi + 1, high);
    }
}

// Linear Search for a school by school_Id
int dream_coders_linear_search(int id) {
    for (int i = 0; i < school_count; i++) {
        if (schools[i].school_Id == id) {
            return i;
        }
    }
    return -1;
}

// Binary Search for a school by school_Id
int dream_coders_binary_search(int id) {
    int low = 0, high = school_count - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (schools[mid].school_Id == id) {
            return mid;
        }
        if (schools[mid].school_Id < id) {
            low = mid + 1;
```

```
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

void dream_coders_create() {
    if (school_count >= max) {
        printf("School List is Full\n");
        return;
    }
    sch s;
    getschooldetails(&s);
    schools[school_count++] = s;
    save_to_file();
    printf("School Created Successfully!\n");
}

void dream_coders_delete() {
    if (school_count == 0) {
        printf("School List not found\n");
        return;
    }
    int id;
    printf("Enter School ID to Delete: ");
    scanf("%d", &id);
    for (int i = 0; i < school_count; i++) {
        if (schools[i].school_Id == id) {
            for (int j = i; j < school_count - 1; j++) {
                schools[j] = schools[j + 1];
            }
            school_count--;
            save_to_file();
            printf("School deleted successfully!\n");
            return;
        }
    }
    printf("School with ID %d not found.\n", id);
}

void dream_coders_update() {
    if (school_count == 0) {
        printf("School List not found\n");
        return;
    }
```

```
    int id;
    printf("Enter School ID to Update: ");
    scanf("%d", &id);
    for (int i = 0; i < school_count; i++) {
        if (schools[i].school_Id == id) {
            printf("Enter details of ID %d again\n", id);
            getschooldetails(&schools[i]);
            save_to_file();
            printf("School updated successfully!\n");
            return;
        }
    }
    printf("School with ID %d not found.\n", id);
}

void dream_coders_retrieve() {
    if (school_count == 0) {
        printf("School List is Empty\n");
        return;
    }
    for (int i = 0; i < school_count; i++) {
        showschooldetails(&schools[i]);
    }
}

void dream_coders_display_time_complexities() {
    printf("\nTime Complexities:\n");
    printf("1. Bubble Sort: O(n^2)\n");
    printf("2. Quick Sort: O(n log n) (on average)\n");
    printf("3. Linear Search: O(n)\n");
    printf("4. Binary Search: O(log n) (requires sorted data)\n");
}

int main() {
    dream_coders_load_from_file();

    int choice;
    do {
        printf("\n--- School Management System ---\n");
        printf("1. Create School\n");
        printf("2. Delete School\n");
        printf("3. Update School\n");
        printf("4. Retrieve School List\n");
        printf("5. Sort Schools (Bubble Sort)\n");
        printf("6. Sort Schools (Quick Sort)\n");
        printf("7. Search School (Linear Search)\n");
```

```
printf("8. Search School (Binary Search)\n");
printf("9. Display Time Complexities\n");
printf("10. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        dream_coders_create();
        break;
    case 2:
        dream_coders_delete();
        break;
    case 3:
        dream_coders_update();
        break;
    case 4:
        dream_coders_retrieve();
        break;
    case 5: {
        clock_t start = clock();
        dream_coders_bubble_sort();
        clock_t end = clock();
        double time_taken = ((double)end - start) / CLOCKS_PER_SEC;
        printf("Bubble Sort Time: %f seconds\n", time_taken);
        break;
    }
    case 6: {
        clock_t start = clock();
        dream_coders_quick_sort(0, school_count - 1);
        clock_t end = clock();
        double time_taken = ((double)end - start) / CLOCKS_PER_SEC;
        printf("Quick Sort Time: %f seconds\n", time_taken);
        break;
    }
    case 7: {
        int id;
        printf("Enter School ID to search: ");
        scanf("%d", &id);
        clock_t start = clock();
        int result = dream_coders_linear_search(id);
        clock_t end = clock();
        double time_taken = ((double)end - start) / CLOCKS_PER_SEC;
        if (result != -1)
            printf("School found at index: %d\n", result);
        else
```

```
                printf("School not found\n");
            printf("Linear Search Time: %f seconds\n", time_taken);
            break;
        }
        case 8: {
            int id;
            printf("Enter School ID to search: ");
            scanf("%d", &id);
            clock_t start = clock();
            int result = dream_coders_binary_search(id);
            clock_t end = clock();
            double time_taken = ((double)end - start) / CLOCKS_PER_SEC;
            if (result != -1)
                printf("School found at index: %d\n", result);
            else
                printf("School not found\n");
            printf("Binary Search Time: %f seconds\n", time_taken);
            break;
        }
        case 9:
            dream_coders_display_time_complexities();
            break;
        case 10:
            save_to_file();
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice, please try again.\n");
        }
    } while (choice != 10);

    return 0;
}
```

Comparison of Sorting Algorithms

BUBBLE SORT

```
#include <iostream>
using namespace std;

    void bubbleSort() {
        sort(students.begin(), students.end(), [](const Student& a, const Student& b) {
            return a.name < b.name;
        });
        cout << "Students sorted by name.\n";
        for (const auto& student : students) {
            student.display();
        }
    }
```

QUICK SORT

```
#include <iostream>
using namespace std;

void dream_coders_quick_sort(int low, int high) {
    if (low < high) {
        int pivot = schools[high].school_Id;
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (schools[j].school_Id < pivot) {
                i++;
                sch temp = schools[i];
                schools[i] = schools[j];
                schools[j] = temp;
            }
        }
        sch temp = schools[i + 1];
        schools[i + 1] = schools[high];
        schools[high] = temp;
        int pi = i + 1;
        dream_coders_quick_sort(low, pi - 1);
        dream_coders_quick_sort(pi + 1, high);
    }
}
```

Difference Between Bubble Sort and Quick Sort

1. Algorithm Type:

- **Bubble Sort**: A simple comparison-based sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It is a quadratic sorting algorithm.
- **Quick Sort**: A divide-and-conquer sorting algorithm that selects a pivot, partitions the array into two subarrays around the pivot, and recursively sorts the subarrays.

2. Time Complexity:

- **Bubble Sort**:
  - Best Case: $O(n)$ (when the array is already sorted).
  - Worst Case: $O(n^2)$ (when the array is sorted in reverse).
- **Quick Sort**:
  - Best Case: $O(n \log n)$ (balanced partitioning).
  - Worst Case: $O(n^2)$ (if the pivot is always the smallest or largest element).
  - Average Case: $O(n \log n)$.

3. Space Complexity:

- **Bubble Sort**: $O(1)$ (in-place sorting, no extra memory needed).
- **Quick Sort**: $O(\log n)$ (for recursion stack in best/average cases).

4. Stability:

- **Bubble Sort**: Stable (maintains the relative order of equal elements).
- **Quick Sort**: Not stable in its basic form.

5. Use Cases:

- **Bubble Sort**: Suitable for small datasets or educational purposes due to its simplicity.
- **Quick Sort**: Ideal for large datasets due to its efficiency and adaptability.

6. Recursion:

- **Bubble Sort**: Non-recursive, iterative algorithm.
- **Quick Sort**: Recursive algorithm.

7. Ease of Implementation:

- **Bubble Sort**: Very simple to implement, beginner-friendly.
- **Quick Sort**: Slightly more complex due to partitioning and recursion.

8. Performance:

- **Bubble Sort**: Inefficient for large datasets due to its quadratic time complexity.

- **Quick Sort**: Efficient for large datasets, especially with good pivot selection.

Key Takeaway:

- Use **Bubble Sort** for small, simple sorting tasks or when learning basic algorithms.
- Use **Quick Sort** for larger datasets or when performance is critical.

Comparison of Searching Algorithms

1.BINARY SEARCH

```
#include <iostream>
using namespace std;

void searchStudent() {
        string name;
        cin.ignore();
        cout << "Enter Student Name to Search: ";
        getline(cin, name);

        int index = binarySearch(name);
        if (index != -1) {
            students[index].display();
        } else {
            cout << "Student not found!\n";
        }
    }

int main() {
    int arr[] = {1, 3, 5, 7, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x;
    cin >> x;

    int result = binarySearch(arr, n, x);
    cout << (result == -1 ? "Not Found" : "Found at index " + to_string(result)) << endl;

    return 0;
}
```

2.LINEAR SEARCH

```
#include <iostream>
using namespace std;

int linearSearch(int arr[], int n, int x) {
    for (int i = 0; i < n; i++)
        if (arr[i] == x) return i;
    return -1;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int x;
    cin >> x;

    int result = linearSearch(arr, n, x);
    cout << (result == -1 ? "Not Found" : "Found at index " + to_string(result)) << endl;


return 0;
}
```

Comparison Search Between Binary Search and Linear

1. Data Requirement:
Linear search works on both sorted and unsorted arrays, while binary search requires the array to be sorted before performing the search.

2. Time Complexity:
Linear search has a time complexity of O(n) because it checks every element one by one until it finds the target or reaches the end of the array. Binary search is more efficient with a time complexity of O(log n) as it divides the search range into halves repeatedly.

3. Space Complexity:
Both iterative binary search and linear search have a space complexity of O(1). However, recursive binary search has a space complexity of O(log n) due to the recursive call stack.

4. Best Case Performance:

In linear search, the best case is when the element is at the beginning of the array, taking only O(1) time.

In binary search, the best case occurs when the target is at the middle index, also taking O(1) time.

5. Worst Case Performance:

Linear search's worst case happens when the element is not in the array or at the very end, taking O(n) time.

Binary search's worst case occurs when the target is not found after repeatedly halving the array, taking O(log n) time.

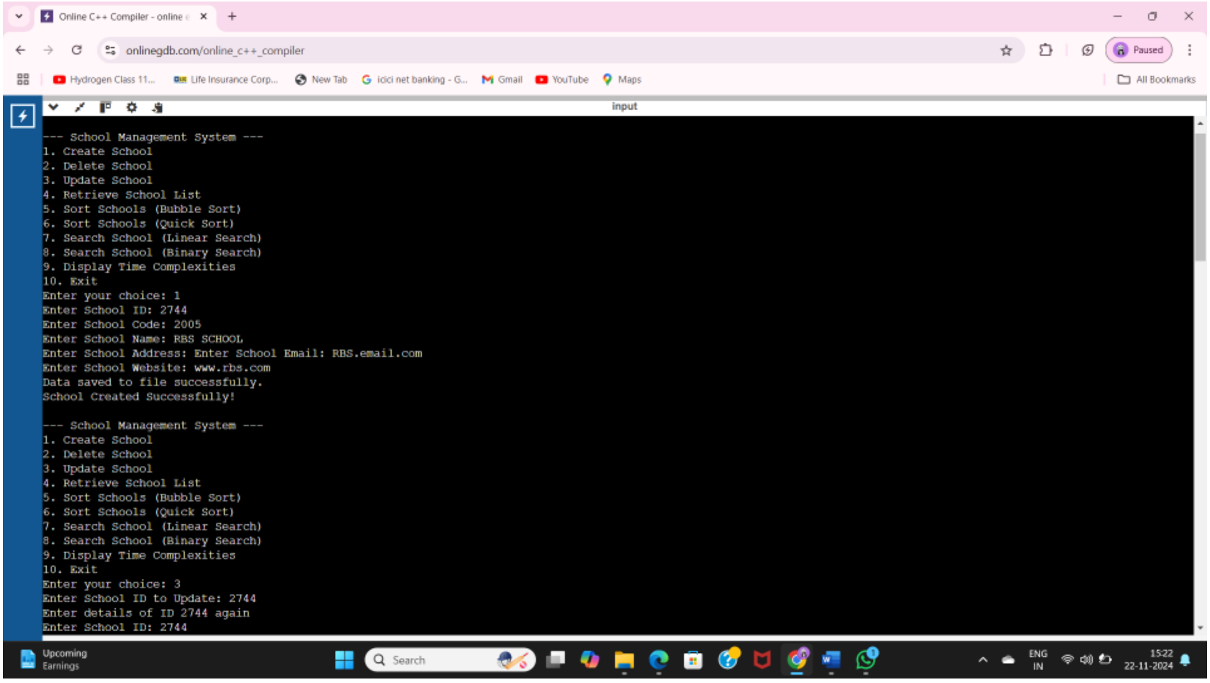
6. Search Mechanism:
Linear search checks every element sequentially, making it straightforward but inefficient for large datasets. Binary search splits the search range in half, drastically reducing the number of comparisons.

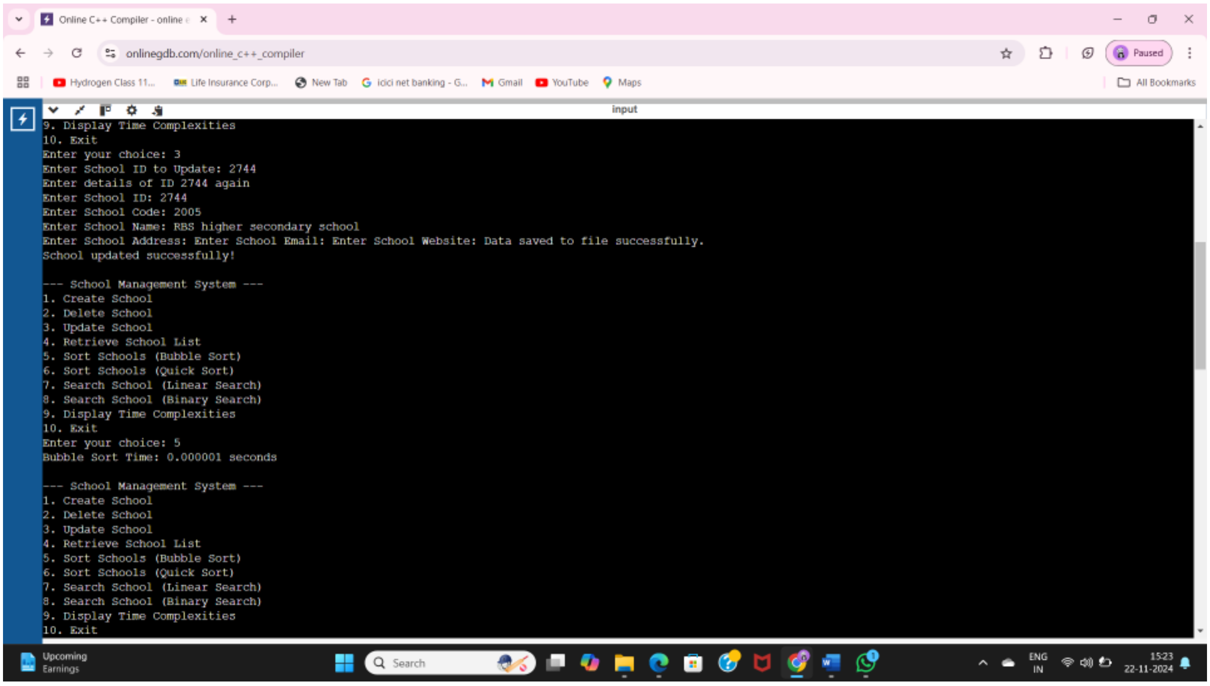
7. Ease of Implementation:
Linear search is simple to implement as it only requires a loop. Binary search is slightly more complex due to the need for calculating the mid-point and handling conditions for sorted data.

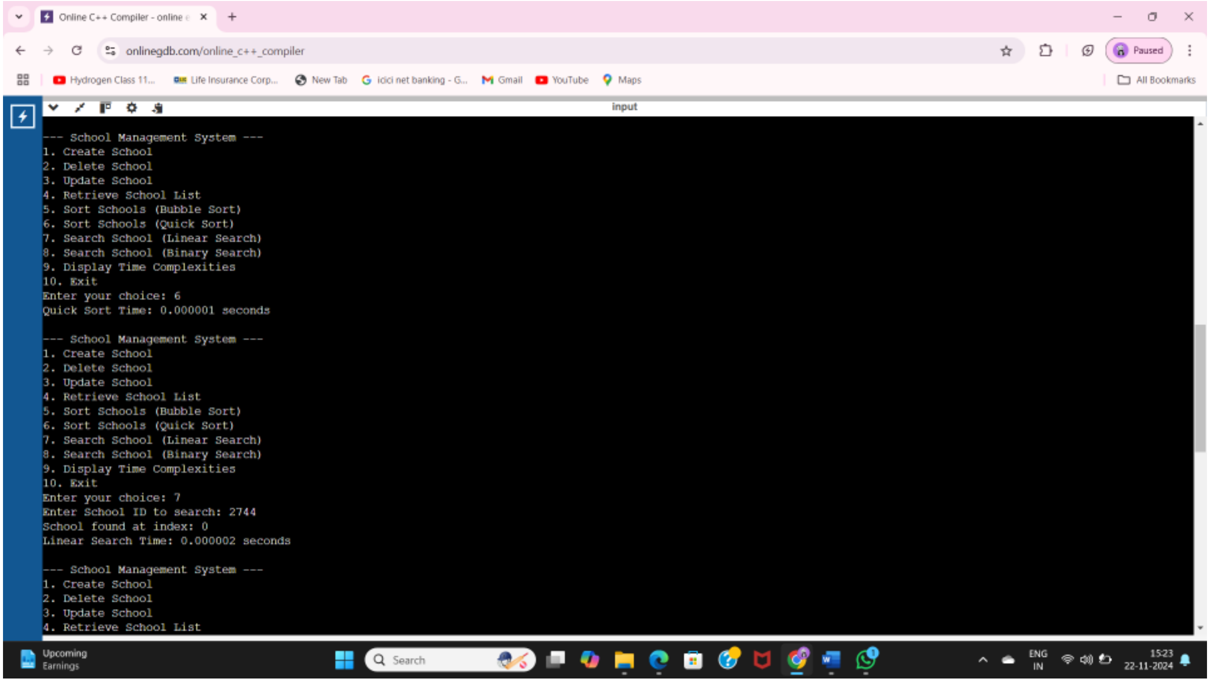
8. Use Cases:
Linear search is suitable for small datasets or when the data isn't sorted.

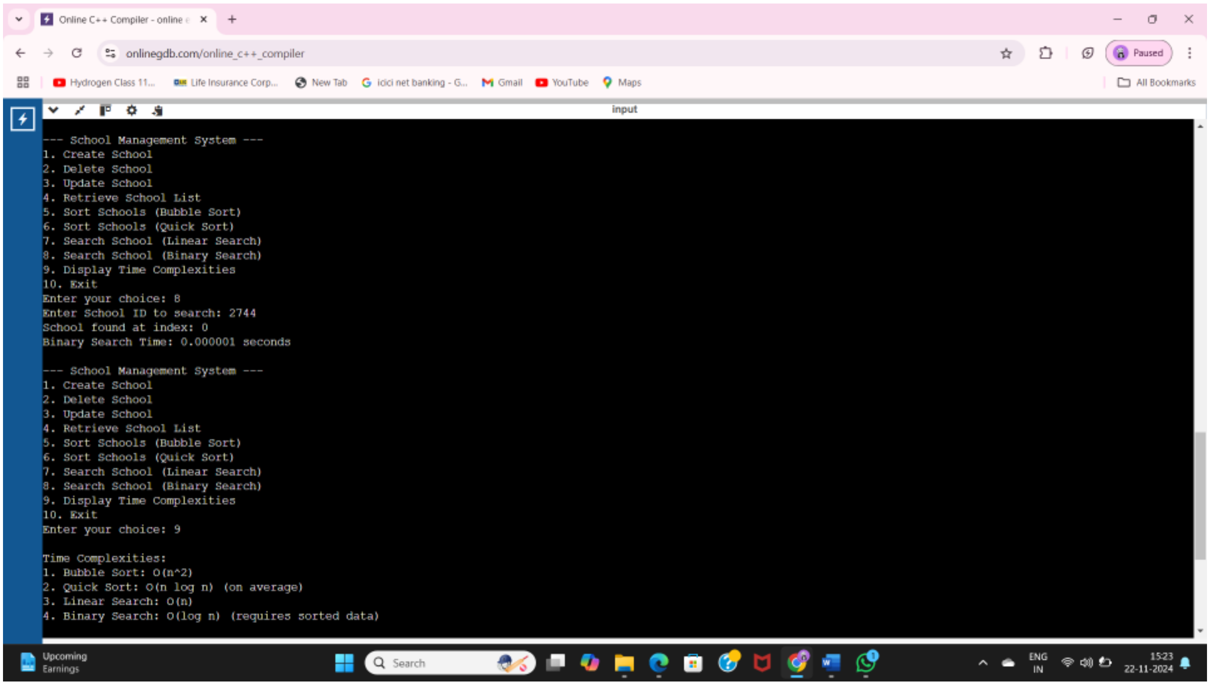
Binary search is ideal for large, sorted datasets and scenarios where multiple searches are needed.

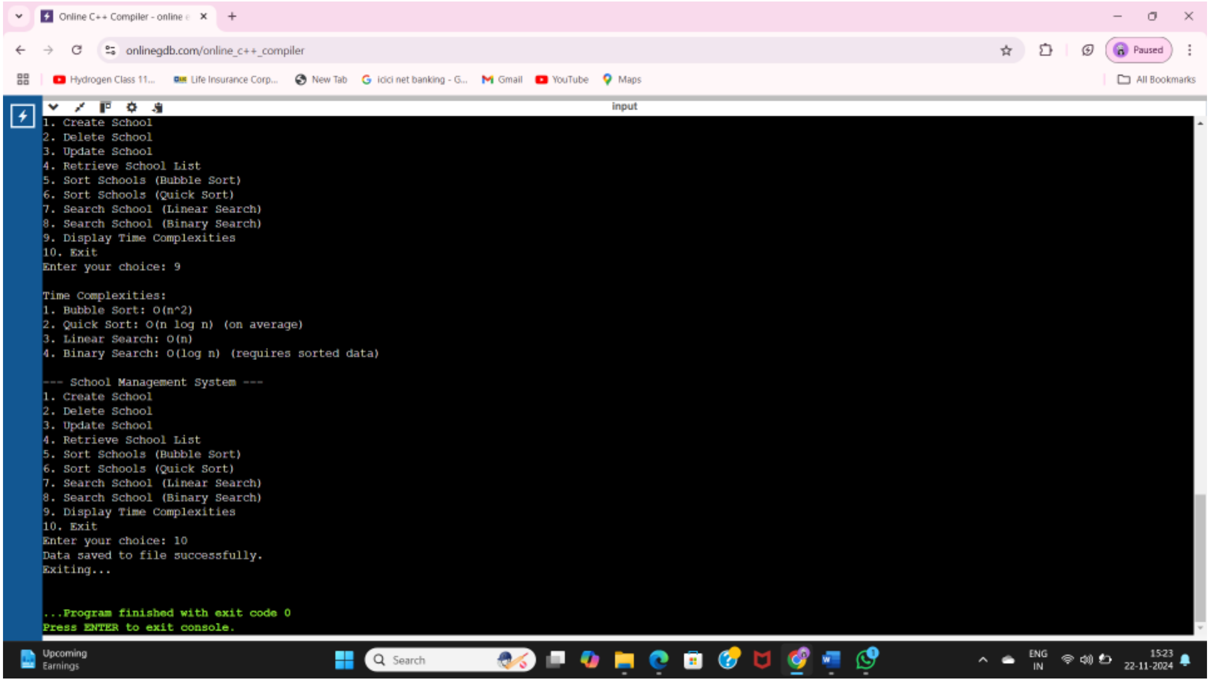
Screen Shots






Online C++ Compiler - online
onlinegdb.com/online_c++_compiler
input
--- School Management System ---
1. Create School
2. Delete School
3. Update School
4. Retrieve School List
5. Sort Schools (Bubble Sort)
6. Sort Schools (Quick Sort)
7. Search School (Linear Search)
8. Search School (Binary Search)
9. Display Time Complexities
10. Exit
Enter your choice: 6
Quick Sort Time: 0.000001 seconds
--- School Management System ---
1. Create School
2. Delete School
3. Update School
4. Retrieve School List
5. Sort Schools (Bubble Sort)
6. Sort Schools (Quick Sort)
7. Search School (Linear Search)
8. Search School (Binary Search)
9. Display Time Complexities
10. Exit
Enter your choice: 7
Enter School ID to search: 2744
School found at index: 0
Linear Search Time: 0.000002 seconds
--- School Management System ---
1. Create School
2. Delete School
3. Update School
4. Retrieve School List
15:23
22-11-2024


Online C++ Compiler - online
onlinegdb.com/online_c++_compiler
input
--- School Management System ---
1. Create School
2. Delete School
3. Update School
4. Retrieve School List
5. Sort Schools (Bubble Sort)
6. Sort Schools (Quick Sort)
7. Search School (Linear Search)
8. Search School (Binary Search)
9. Display Time Complexities
10. Exit
Enter your choice: 8
Enter School ID to search: 2744
School found at index: 0
Binary Search Time: 0.000001 seconds
--- School Management System ---
1. Create School
2. Delete School
3. Update School
4. Retrieve School List
5. Sort Schools (Bubble Sort)
6. Sort Schools (Quick Sort)
7. Search School (Linear Search)
8. Search School (Binary Search)
9. Display Time Complexities
10. Exit
Enter your choice: 9
Time Complexities:
1. Bubble Sort: O(n^2)
2. Quick Sort: O(n log n) (on average)
3. Linear Search: O(n)
4. Binary Search: O(log n) (requires sorted data)
15:23
22-11-2024


Online C++ Compiler - online
onlinegdb.com/online_c++_compiler
input
1. Create School
2. Delete School
3. Update School
4. Retrieve School List
5. Sort Schools (Bubble Sort)
6. Sort Schools (Quick Sort)
7. Search School (Linear Search)
8. Search School (Binary Search)
9. Display Time Complexities
10. Exit
Enter your choice: 9
Time Complexities:
1. Bubble Sort: O(n^2)
2. Quick Sort: O(n log n) (on average)
3. Linear Search: O(n)
4. Binary Search: O(log n) (requires sorted data)
--- School Management System ---
1. Create School
2. Delete School
3. Update School
4. Retrieve School List
5. Sort Schools (Bubble Sort)
6. Sort Schools (Quick Sort)
7. Search School (Linear Search)
8. Search School (Binary Search)
9. Display Time Complexities
10. Exit
Enter your choice: 10
Data saved to file successfully.
Exiting...
...Program finished with exit code 0
Press ENTER to exit console.

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.