OBE IMPLEMENTATION – UNIVERSITY SETTING

Subtitle: Outcome-Based Education Project
Submitted By: S.Harshith Kumar
Reg No.: AP23110010709
Department: CSE
Semester: 3rd Semester
Regulation: [Your Regulation]
Date: 14-11-2024

TEAM MEMBERS

HARSHITH	AP23110010709
ANIMESH	AP23110011096
ARJUN	AP23110010706
RISHI	AP23110011088
PRRANET	AP23110010713
FAIZAN ALI	AP23110011107

INTRODUCTION TO PROJECT

Objective:
The project is part of the university's initiative to implement Outcome-Based Education (OBE). The primary goal is to design an application for managing university records, implementing CRUD operations (Create, Retrieve, Update, Delete), and performing sorting and searching using algorithms.

Key Features:
CRUD Operations: Users can create, update, retrieve, and delete university records.
Search and Sort: Sorting and searching algorithms will be implemented to efficiently manage and find records.

Time Complexity: Time complexity analysis for both sorting and searching algorithms.
File Handling: Data will be stored in a text file (university_setting.txt) for persistence across sessions.

Programming Language: The application is developed using C programming language.

Outcome: The project will allow users to manage and retrieve data in an efficient manner using basic sorting and searching algorithms, with a focus on performance and complexity analysis.

ARCHITECTURE DIAGRAM

Components:
User Interface (CLI): A command-line interface (CLI) allows the user to interact with the system t
perform various actions like adding, updating, retrieving, and deleting university records.
University Data Storage: All university records are stored in a text file (university_setting.txt).

CRUD Operations: The system supports basic CRUD functionality.
Sorting and Searching: Sorting algorithms (Insertion Sort, Selection Sort) and searching algorithm
(Binary Search, Linear Search) allow efficient data organization and retrieval.

Time Complexity Calculation: The system will calculate and display the time complexity o
both sorting and searching algorithms.
Note: Highlight the central module, which is the University Setting system that handles a
operations.

MODULE DESCRIPTION – UNIVERSITY SETTING

Purpose: This module is designed to manage university records, supporting four key operations:

Create: Add new university records.

Retrieve: Display the current list of university records.

Update: Modify existing records.

Delete: Remove a university record from the system.

Search and Sort: Search for universities based on university code and sort records in lexicographical order.

Create: Adds new universities with details like university code, name, address, email, and website.

Retrieve: Lists all the universities stored.

Update: Allows updating of a university's details.

Delete: Removes a university's record based on the university ID.

University Setting - Field/Table Details

University Setting Fields:

Field Name	Data Type
id	integer
univ_code	string
univ_name	string
univ_address	string
univ_email	string
univ_website	string

Explanation:
id: A unique identifier for each university.
univ_code: A string representing the university's code (e.g., SRM-AP).
univ_name: The name of the university.
univ_address: The university's physical address.
univ_email: Contact email for the university.
univ_website: The official website URL for the university.

PROGRAMMING DETAILS - FILE AND FUNCTION NAMES

File Name:

AP23110011096_university_setting.c

This file contains all the functions and logic related to CRUD operations, sorting, searching, and file storage.

Function Names:

Create: AP23110011096_UNIVERSITY_create()
Update: AP23110011096_UNIVERSITY_update()
Retrieve: AP23110011096_UNIVERSITY_retrieve()
Delete: AP23110011096_UNIVERSITY_delete()

Sorting:

AP23110011096_UNIVERSITY_INSERTION_SORT()
AP23110011096_UNIVERSITY_SELECTION_SORT()

Searching:

AP23110011096_UNIVERSITY_LINEAR_SEARCH()
AP23110011096_UNIVERSITY_BINARY_SEARCH()

Storing: AP23110011096_UNIVERSITY_storing()

COMPARISON OF ALGORITHMS

For Searching:
AP23110011096_UNIVERSITY_Compare_Search()

Time Complexity: AP23110011096_UNIVERSITY_complexity_Search()

For Sorting:
AP23110011096_UNIVERSITY_Compare_sorting()

Time Complexity: AP23110011096_UNIVERSITY_complexity_sorting()

Key Comparison Points:

Linear Search vs. Binary Search:
Linear Search: Searches element by element (O(n) time complexity).
Binary Search: Requires sorted data and divides the search space by half each time (O(log n) time complexity).

Insertion Sort vs. Selection Sort:
Insertion Sort: Builds the sorted array element by element (best case O(n), worst case $O(n^2)$).
Selection Sort: Finds the minimum element and swaps ($O(n^2)$ in all cases).

ALGORITHM DETAILS - SEARCHING (LINEAR SEARCH)

Pseudocode/Steps:

Prompt the user for the univ_code they wish to find.
Loop through each element in the universities array.
Compare each element's univ_code with the search univ_code.

If a match is found, print the university details and end the search.

If no match is found after searching all elements, print a "not found" message.
Time Complexity:
Best Case: O(1) (if the element is at the start of the list).
Worst Case: O(n) (if the element is at the end or not present).

ALGORITHM DETAILS – SEARCHING (BINARY SEARCH)

Pseudocode/Steps:

Ensure the universities array is sorted by univ_code.

Set two pointers: low to 0 and high to university_count - 1.

Repeat the following steps while low is less than or equal to high:

Calculate the midpoint: mid = (low + high) / 2.

If universities[mid].univ_code matches the search univ_code, print the details and end the search.

If universities[mid].univ_code is less than the search univ_code, set low = mid + 1.

If universities[mid].univ_code is greater than the search univ_code, set high = mid - 1.

If no match is found, print a “not found” message.

Time Complexity:

Best Case: O(1) (if the middle element is the match).

Worst and Average Case: O(log n) (due to halving the search range with each iteration).

SORTING ALGORITHM – INSERTION SORT

Pseudocode/Steps:

Start from the second element (i = 1).

Store the current university record (universities[i]) in a temporary variable (temp).

Set j to i - 1.

Compare temp.univ_code with universities[j].univ_code.

Shift universities[j] to universities[j + 1] if universities[j].univ_code is greater than temp.univ_code.

Continue shifting until temp.univ_code is in the correct position.

Insert temp at the correct position in the array.

Repeat steps 2–7 for all elements in the array.

Time Complexity:

Best Case: O(n) (if the array is already sorted).

Worst Case: $O(n^2)$ (if the array is sorted in reverse order).

SORTING ALGORITHM – SELECTION SORT

Pseudocode/Steps:

Loop over each element of the array (i from 0 to university_count - 1).

Assume the current element (i) is the minimum.

Loop through the unsorted part of the array (j from i + 1 to university_count).

Update the minimum index if a smaller univ_code is found.

Swap the current element (universities[i]) with the minimum element found in the unsorted part.

Repeat until the array is fully sorted.

Time Complexity:

Best, Average, and Worst Case: $O(n^2)$

TIME COMPLEXITY OF SORTING ALGORITHMS

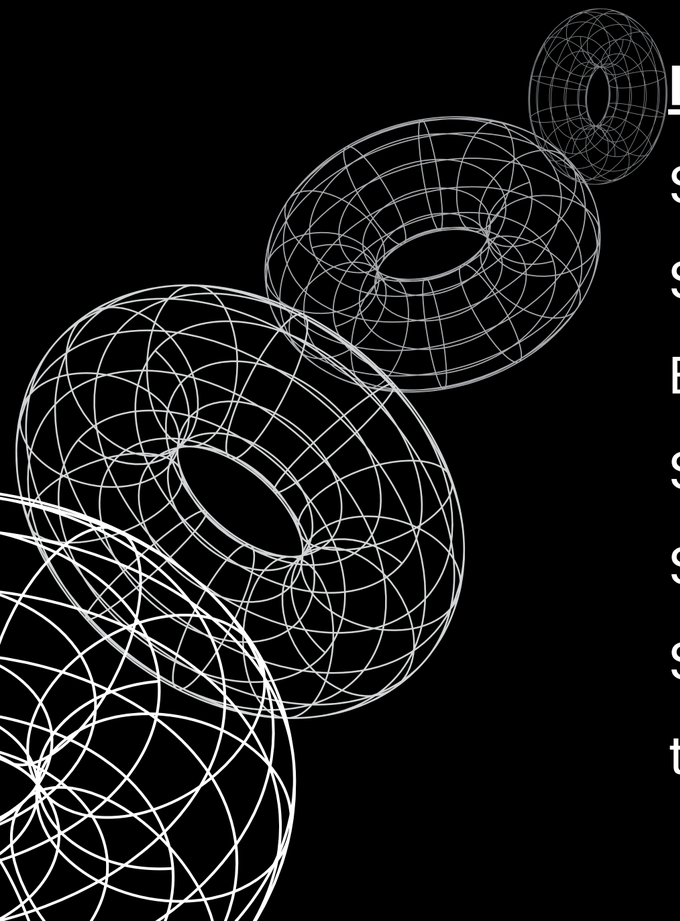
Time Complexity Table

Sl. No.	Algorithm Name	Compared Algorithm	Time Complexity
1	Insertion Sort	Selection Sort	$O(n)$, $O(n^2)$
2	Binary Search	Linear Search	$O(\log n)$, $O(n)$

I**NSERTION SORT:** BEST CASE: O(N), WORST CASE: O(N²)

SELECTION SORT: O(N²) IN ALL CASES

SEARCHING ALGORITHM - COMPARISON (LINEAR VS. BINARY SEARCH)



Linear Search:

Step-1: Compare the target with each element sequentially.

Step-2: Continue until a match is found or the list ends.

Binary Search:

Step-1: Ensure the array is sorted.

Step-2: Compare the middle element with the target.

Step-3: If it matches, return the index. If not, search either the left or right half.

TIME COMPLEXITY OF SEARCHING ALGORITHMS

Sl. No	Algorithm Name	Compared Algorithm	Time Complexity
1	Linear Search	Binary Search	O(n)
2	Binary Search	Linear Search	O(log n)

SOURCE CODE

HTTPS://GITHUB.COM/ANIMESHCSE47/DAA-PROJECT

CONCLUSION

Project Outcome:

The system is capable of managing university data using basic CRUD operations.

Sorting and searching operations are implemented efficiently.

The project demonstrates fundamental algorithm analysis and helps understand the complexities of sorting and searching algorithms in C.

Future Improvements:

Add a graphical user interface (GUI) for ease of use.

Implement advanced searching techniques (e.g., hashing).

THANK YOU