OBE IMPLEMENTATION:

UNIVERSITY SETTING

BY

Arjun Awasthi (AP23110010706)

Harshith Kumar (AP23110010709)

Animesh (AP23110011096)

Rishi Kumar (AP23110011088)

Prraneth Araveti (AP23110010713)

Faizaan Reza (AP23110011107)

A report for the Design and Analysis of Algorithm (CSE-204) project.



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**SRM UNIVERSITY AP :: AMARAVATI**

INDEX

* Introduction

(i)Project Modules

* Architecture Diagram
* Module Description
* Programming Details naming conventions to be used:
* Field/table details
* Algorithm Details:

1. Sorting
2. Searching

(iii) Storing the details in a text file

* Source Code
* Comparison of Sorting Algorithms
* Comparison of Searching Algorithms
* Screen Shots
* Conclusion

INTRODUCTION

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.

PROJECT MODULES:

1. University Information Management Module

Description: This module handles all CRUD operations (Create, Retrieve, Update, Delete) for storing and managing data related to universities. Key fields include university code, name, address, email, and website.

Functions: Add new universities, view existing records, update details, delete records, store data in university\_setting.txt.

File Management: Automatically saves changes to the text file and updates entries as per user modifications.

2. Student Outcome Tracking Module

Description: This module records and tracks student learning outcomes, mapping them to defined OBE criteria. It provides options for entering and viewing outcomes related to specific students.

Functions: Add, view, and update student outcome records, generate summaries of achieved learning outcomes, and link outcomes to student IDs and programs.

3. Curriculum Mapping and Assessment Module

Description: This module maps course outcomes to the curriculum, ensuring each course's learning objectives align with OBE goals. It enables administrators to link course content with specific learning outcomes.

Functions: Map course outcomes to curriculum objectives, store course data (title, outcomes, faculty assigned), and link curriculum components to measurable OBE criteria.

4. Faculty and Course Evaluation Module

Description: Provides tools for evaluating faculty performance and course effectiveness. The module measures faculty performance based on student outcomes and evaluates how well course content contributes to intended learning results.

Functions: Collect and store faculty assessments, calculate evaluation metrics, and generate performance reports. Also provides feedback on course effectiveness related to OBE standards.

5. Performance Analysis and Reporting Module

Description: Generates reports for students, faculty, and administrators based on collected data. It includes detailed analysis of outcomes, course effectiveness, and faculty performance to help improve the OBE process.

Functions: Generate performance summaries and detailed reports, provide insights for continuous improvement, and track key performance indicators (KPIs) for students and faculty.

6. Data Sorting and Searching Module

Functions: Sort records by various fields (e.g., university name, code), and search based on criteria like university email or student outcome. Algorithms include Quick Sort, Merge Sort, Binary Search, and Linear Search.

7. Data Storage and Backup Module

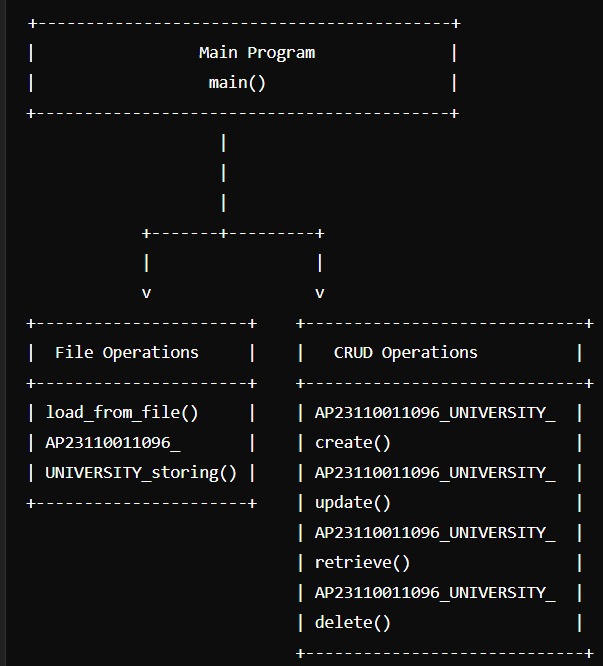
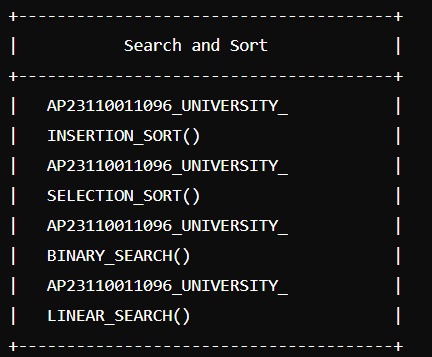
Description: Manages data storage within the text file system and ensures that data entries are saved, updated, and deleted as per CRUD operations. This module also offers backup and restoration features for data security.

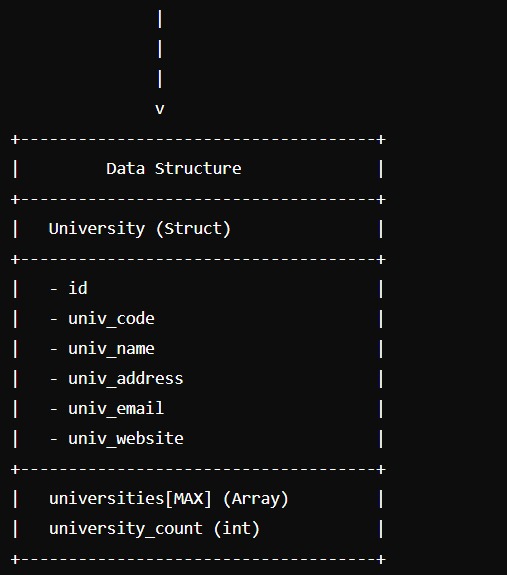
Functions: Save data to university\_setting.txt, ensure data persistence for CRUD operations, and maintain a backup copy to prevent data loss.

8. Algorithm Comparison Module

Description: Compares the efficiency of different sorting and searching algorithms implemented within the project. This module provides insights on performance, including time complexity and resource utilization.

Functions: Run algorithm comparisons for sorting (e.g., Quick Sort vs. Merge Sort) and searching (e.g., Binary Search vs. Linear Search), display time complexities, and show pseudocode for each algorithm.

ARCHITECTURE DIAGRAM



MODULE DESCRIPTION

* Module Name: University Setting
* Module Description:

This module manages data about various universities, providing essential CRUD (Create, Retrieve, Update, Delete) functionalities. The goal is to allow users to seamlessly add, view, modify, or remove university details in a structured way, ensuring data is stored in an accessible format. The module also includes sorting and searching features for effective data retrieval based on key university attributes.

* Programming Details and Naming Conventions:

To maintain consistency across the module, specific naming conventions are adhered to, based on the registration number.

* File Name: AP23110010706\_university\_setting
* Function/Method Names:

Create Function: AP23110010706\_university\_create ()

Update Function: AP23110010706\_university\_update ()

Retrieve Function: AP23110010706\_university\_retrieve ()

Delete Function: AP23110010706\_university\_delete()

Sorting Function: AP23110010706\_university\_selection\_sort()

Sorting Function: AP23110010706\_university\_insertion\_sort()

Searching Function: AP23110010706\_university\_linear\_search ()

Searching Function: AP23110010706\_university\_binary\_search ()

Storing Function: AP23110010706\_university\_storing

* Comparison:

Searching Comparison: AP23110010706\_university\_Compare\_Search\_selection\_sort

b/w

AP23110010706\_university\_Compare\_Search\_insertion\_sort

Sorting Comparison: AP23110010706\_university\_Compare\_sorting\_linear\_search

b/w

AP23110010706\_university\_Compare\_sorting\_binary\_search

* Time Complexity Functions:

Sorting Complexity:AP23110010706\_university\_complexity\_search

Sorting Complexity:AP23110010706\_university\_complexity\_sorting

* Field/Table Details:

|  |  |
| --- | --- |
| Field Name | Data Type |
| Id | Integer |
| univ\_code | String |
| univ\_name | String |
| univ\_address | String |
| univ\_email | String |
| univ\_website | String |

* Algorithm Details:

1. Sorting:

Sorting operations are implemented on fields such as univ\_code, univ\_name, and univ\_email.

Primary Algorithm:

Insertion Sort:

Time Complexity:

𝑂(𝑛^2) in the average and worst cases and

O(n) in the best case (when the data is nearly sorted).

Sorting Based on: univ\_code, univ\_name, univ\_email.

Comparison Algorithm:

Time Complexity:

O(n^2) in all cases.

Pseudocode:

Insertion Sort builds a sorted array one element at a time by inserting each element into its correct position within the already sorted section.

function insertion\_Sort(arr)

for i=1 to length(arr) -1

key = arr[i]

j = i - 1

while j >= 0 and arr[j] > key

arr[j+1] = arr[j]

j = j -1

arr[j+1] = key

Selection Sort:

Selection Sort repeatedly finds the minimum element from the unsorted part and moves it to the sorted part of the array.

function selectionSort(arr)

for i = 0 to length(arr) - 1

minIndex = i

for j = i + 1 to length(arr) - 1

if arr[j] < arr[minIndex]

minIndex = j

swap arr[i] and arr[minIndex]

* Updated Comparison Functions:

Sorting Comparison: The comparison function would compare Insertion Sort with Selection Sort, highlighting the performance differences in various scenarios.

(ii) Searching :

Searching Based on: univ\_code, univ\_name, univ\_email.

Algorithm Used: Binary Search

Comparison Algorithm: Linear Search

Time Complexity:

Binary Search: O(log n)

Linear Search: O(n)

Pseudocode for Binary Search:

function binarySearch(arr, target)

low = 0

high = length of arr - 1

while low <= high

mid = (low + high) / 2

if arr[mid] == target

return mid

else if arr[mid] < target

low = mid + 1

else

high = mid - 1

return –

(iii)Storing Details in a Text File :

File Name: university\_setting.txt

Storage Operations:

Add: When a new university record is created, it is appended to the text file.

Update: When any detail is modified, the file is re-saved with updated information.

Delete: When a record is deleted, it is removed from the text file, and the remaining records are saved back to the file.

SOURCE CODE

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

// University structure

typedef struct {

int id;

char univ\_code[10];

char univ\_name[50];

char univ\_address[100];

char univ\_email[50];

char univ\_website[50];

} University;

// Global array to store university data and a counter

University universities [MAX];

int university\_count = 0;

// File name for storing the details

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

// Function declarations

void AP23110010706\_UNIVERSITY\_create();

void AP23110010706\_UNIVERSITY\_update();

void AP23110010706\_UNIVERSITY\_retrieve();

void AP23110010706\_UNIVERSITY\_delete();

void AP23110010706UNIVERSITY\_storing();

void AP23110010706\_UNIVERSITY\_INSERTION\_SORT();

void AP23110010706\_UNIVERSITY\_SELECTION\_SORT();

void AP23110010706\_UNIVERSITY\_BINARY\_SEARCH();

void AP23110010706\_UNIVERSITY\_LINEAR\_SEARCH();

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

void load\_from\_file() {

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

if (file == NULL) {

return; // No file exists yet

}

university\_count = 0;

while (fscanf(file, "%d %s %s %s %s %s\n", &universities[university\_count].id,

universities[university\_count].univ\_code,

universities[university\_count].univ\_name,

universities[university\_count].univ\_address,

universities[university\_count].univ\_email,

universities[university\_count].univ\_website) != EOF) {

university\_count++;

}

fclose(file);

}

// Function to save data to the file

void AP23110010706\_UNIVERSITY\_storing() {

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

if (file == NULL) {

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

return;

}

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

fprintf(file, "%d %s %s %s %s %s\n", universities[i].id,

universities[i].univ\_code, universities[i].univ\_name,

universities[i].univ\_address, universities[i].univ\_email,

universities[i].univ\_website);

}

fclose(file);

}

// Function to create a university record

void AP23110010706\_UNIVERSITY\_create() {

if (university\_count >= MAX) {

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

return; }

University u;

printf("Enter University ID: ");

scanf("%d", &u.id);

printf("Enter University Code: ");

scanf("%s", u.univ\_code);

printf("Enter University Name: ");

scanf("%s", u.univ\_name);

printf("Enter University Address: ");

scanf("%s", u.univ\_address);

printf("Enter University Email: ");

scanf("%s", u.univ\_email);

printf("Enter University Website: ");

scanf("%s", u.univ\_website);

universities[university\_count++] = u;

AP23110010706\_UNIVERSITY\_storing();

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

}

// Function to update a university record

void AP23110010706\_UNIVERSITY\_update() {

int id;

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

scanf("%d", &id);

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

if (universities[i].id == id) {

printf("Enter new University Code: ");

scanf("%s", universities[i].univ\_code);

printf("Enter new University Name: ");

scanf("%s", universities[i].univ\_name);

printf("Enter new University Address: ");

scanf("%s", universities[i].univ\_address);

printf("Enter new University Email: ");

scanf("%s", universities[i].univ\_email);

printf("Enter new University Website: ");

scanf("%s", universities[i].univ\_website);

AP23110010706\_UNIVERSITY\_storing();

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

return;

}

}

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

}

// Function to retrieve all university records

void AP23110010706\_UNIVERSITY\_retrieve() {

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

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

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

universities[i].id, universities[i].univ\_code,

universities[i].univ\_name, universities[i].univ\_address,

universities[i].univ\_email, universities[i].univ\_website);

}

}

// Function to delete a university record

void AP23110010706\_UNIVERSITY\_delete() {

int id;

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

scanf("%d", &id);

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

if (universities[i].id == id) {

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

universities[j] = universities[j + 1];

}

university\_count--;

AP23110010706\_UNIVERSITY\_storing();

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

return;

}

}

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

}

// Insertion sort by university code

void AP23110010706\_UNIVERSITY\_INSERTION\_SORT() {

for (int i = 1; i < university\_count; i++) {

University key = universities[i];

int j = i - 1;

while (j >= 0 && strcmp(universities[j].univ\_code, key.univ\_code) > 0) {

universities[j + 1] = universities[j];

j = j - 1;

}

universities[j + 1] = key;

}

printf("Universities sorted by code (Insertion Sort)!\n");

AP23110010706\_UNIVERSITY\_retrieve();

}

// Selection sort by university code

void AP23110010706\_UNIVERSITY\_SELECTION\_SORT() {

for (int i = 0; i < university\_count - 1; i++) {

int min\_idx = i;

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

if (strcmp(universities[j].univ\_code, universities[min\_idx].univ\_code) < 0) {

min\_idx = j;

}

}

University temp = universities[min\_idx];

universities[min\_idx] = universities[i];

universities[i] = temp;

}

printf("Universities sorted by code (Selection Sort)!\n");

AP23110010706\_UNIVERSITY\_retrieve();

}

// Binary search for university by code

void AP23110010706\_UNIVERSITY\_BINARY\_SEARCH() {

char code[10];

printf("Enter University Code to search (Binary Search): ");

scanf("%s", code);

int left = 0, right = university\_count - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int res = strcmp(universities[mid].univ\_code, code);

if (res == 0) {

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

universities[mid].id, universities[mid].univ\_code,

universities[mid].univ\_name, universities[mid].univ\_address,

universities[mid].univ\_email, universities[mid].univ\_website);

return;

}

if (res < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

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

}

// Linear search for university by code

void AP23110010706\_UNIVERSITY\_LINEAR\_SEARCH() {

char code[10];

printf("Enter University Code to search (Linear Search): ");

scanf("%s", code);

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

if (strcmp(universities[i].univ\_code, code) == 0) {

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

universities[i].id, universities[i].univ\_code,

universities[i].univ\_name, universities[i].univ\_address,

universities[i].univ\_email, universities[i].univ\_website);

return;

}

}

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

int main() {

load\_from\_file();

int choice;

while (1) {

printf("\n1. Create University\n2. Update University\n3. Retrieve Universities\n4. Delete University\n5. Linear Search by Code\n6. Binary Search by Code\n7. Insertion Sort by Code\n8. Selection Sort by Code\n9. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: AP23110010706\_UNIVERSITY\_create(); break;

case 2: AP23110010706\_UNIVERSITY\_update(); break;

case 3: AP23110010706\_UNIVERSITY\_retrieve(); break;

case 4: AP23110010706\_UNIVERSITY\_delete(); break;

case 5: AP23110010706\_UNIVERSITY\_LINEAR\_SEARCH(); break;

case 6: AP23110010706\_UNIVERSITY\_BINARY\_SEARCH(); break;

case 7: AP23110010706\_UNIVERSITY\_INSERTION\_SORT(); break;

case 8: AP23110010706\_UNIVERSITY\_SELECTION\_SORT(); break;

case 9: exit(0);

default: printf("Invalid choice!\n"); break;

}

}

return 0;

}

COMPARISION OF SORTING ALGORITHMS

**Sorting Algorithms**

**1. Insertion Sort (Function: AP2311001**0706**\_UNIVERSITY\_INSERTION\_SORT)**

* **Purpose**: Sort the universities array by univ\_code in lexicographical order.
* **Algorithm**:
  1. Start from the second element (i = 1) in the universities array.
  2. Store the current university record (universities[i]) in a temporary variable (temp).
  3. Set j to i - 1.
  4. Compare temp.univ\_code with universities[j].univ\_code.
  5. Shift universities[j] to universities[j + 1] if universities[j].univ\_code is greater than temp.univ\_code.
  6. Continue shifting until temp.univ\_code is in the correct position.
  7. Insert temp at the correct position in the array.
  8. Repeat steps 2–7 for all elements in the array.
* **Time Complexity**:
  1. Best Case: O(n)O(n)O(n) when the array is already sorted.
  2. Average and Worst Case: O(n2)O(n^2)O(n2).

**2. Selection Sort (Function: AP2311001**0706**\_UNIVERSITY\_SELECTION\_SORT)**

* **Purpose**: Sort the universities array by univ\_code in lexicographical order.
* **Algorithm**:
  1. Loop over each element of the array (i from 0 to university\_count - 1).
  2. Assume the current element (i) is the minimum.
  3. Loop through the unsorted part of the array (j from i + 1 to university\_count).
  4. Update the minimum index if a smaller univ\_code is found.
  5. Swap the current element (universities[i]) with the minimum element found in the unsorted part.
  6. Repeat until the array is fully sorted.
* **Time Complexity**:
  1. Best, Average, and Worst Case: O(n2)O(n^2)O(n2), since each element is compared with every other element.

COMPARISION OF SEARCHING ALGORITHMS

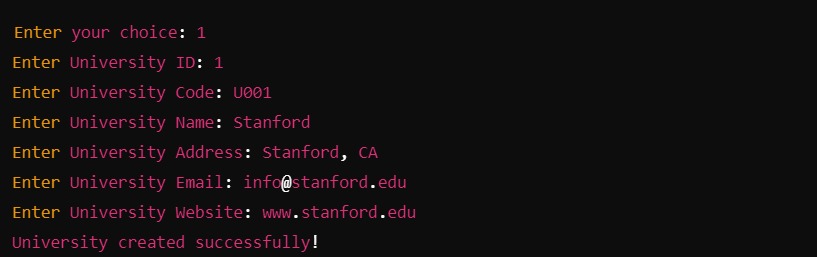
**1. Linear Search (Function: AP2311001**0706**\_UNIVERSITY\_LINEAR\_SEARCH)**

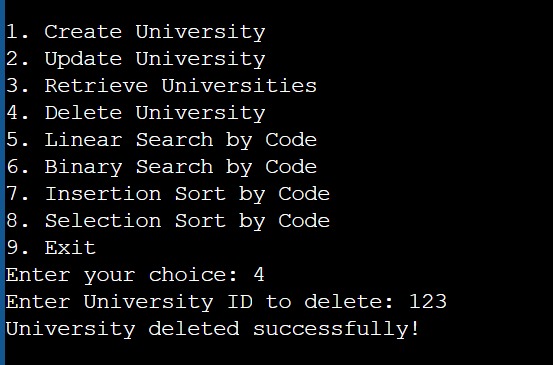
* **Purpose**: Search for a university by univ\_code.
* **Algorithm**:
  1. Prompt the user for the univ\_code they wish to find.
  2. Loop through each element in the universities array.
  3. Compare each element’s univ\_code with the search univ\_code.
  4. If a match is found, print the university details and end the search.
  5. If no match is found after searching all elements, print a “not found” message.
* **Time Complexity**:
  1. Best Case: O(1)O(1)O(1) if the item is at the start of the list.
  2. Worst Case: O(n)O(n)O(n) if the item is at the end or not present.

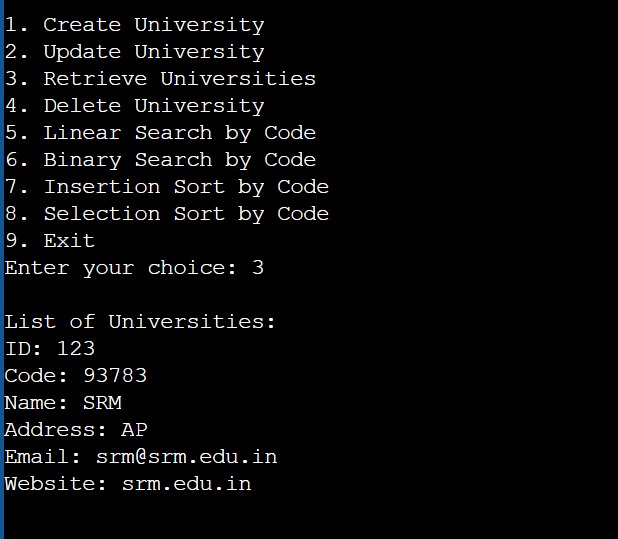
**2. Binary Search (Function: AP2311001**0706**\_UNIVERSITY\_BINARY\_SEARCH)**

* **Purpose**: Search for a university by univ\_code in a sorted array.
* **Algorithm**:
  1. Ensure the universities array is sorted by univ\_code.
  2. Set two pointers: low to 0 and high to university\_count - 1.
  3. 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.
  4. If no match is found, print a “not found” message.
* **Time Complexity**:
  1. Best Case: O(1)O(1)O(1) if the middle element is the match.
  2. Worst and Average Case: O(log⁡n)O(\log n)O(logn) due to dividing the search range in half each time.

SCREENSHOTS(OUTPUT)







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).