**OBE IMPLEMENTATION: COURSES**

By

**Team Name:Executives**

G. Madhu Swapnika (AP23110011011)

M. Abhighna (AP23110011051)

V. Nikitha (AP23110011062)

B. Losish Ram (AP23110010703)

M. Akhil Sai (AP23110011013)

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



DEPARTMENTOFCOMPUTERSCIENCEANDENGINEERING **SRMUNIVERSITYAP:AMARAVAT**I

**INDEX**

**Introduction**

**Modules in the project:**

**Architecture Diagram**

Instructions:

**Module Description**

Programming Details naming conventions to be used:

Field/table details : Courses

Algorithm Details:

(i)Sorting

(ii)Searching

(iii) Storing the details in a text file 6

**Sample coding Template**

OB\_.MAIN\_EXECUTIVES.c

Instruction to use Sample Code:

**ChatGPT Usage**

Program Generated by ChatGPT

Instruction to use ChatGPT or other LLM Models

### Introduction to Courses in C++

The Course Management System (CMS) in C++ is designed to manage and organize course-related information efficiently. This system allows administrators or educators to perform various operations such as adding new courses, updating existing ones, retrieving detailed course information, and sorting courses by specific attributes like course code. The system also supports efficient searching techniques, allowing users to quickly find a course based on a given code. The core functionality includes the creation, storage, retrieval, searching, and sorting of courses using algorithms such as binary search and merge sort, ensuring scalability and performance.

### Architecture Diagram of Course and Module Description

Below is the simplified architecture for the Course Management System,

+------------------------------------+

| |

+------------------------------------+

| |

| +-----------------------------+ |

| | Course | |

| +-----------------------------+ |

| | - course\_id | |

| | - program\_id | |

| | - course\_code | |

| | - course\_name | |

| | - course\_category | |

| | - course\_type | |

| | - prerequisite | |

| | - corequisite | |

| | - progression | |

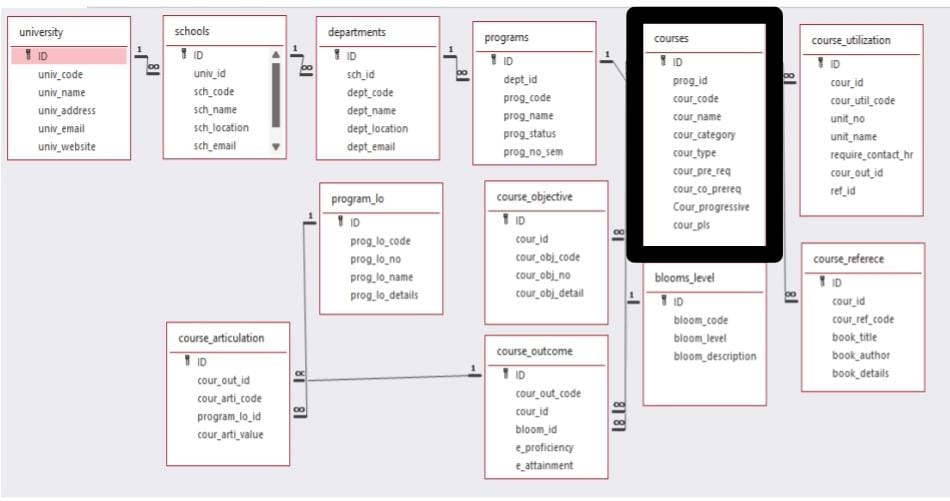
| | - pls (program learning stmt) | |

| +-----------------------------+ |

| |

+------------------------------------+

# Architecture Diagram:



.

### Module Descriptions:

**Module Name:** *Course*

**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

1. **Course Structure**: Each course in the system is represented by a Course structure (similar to the Executive structure in the provided code). It contains important fields such as:
   * course\_id: A unique identifier for the course.
   * program\_id: The program to which the course belongs.
   * course\_code: The unique code used to identify the course.
   * course\_name: The name of the course.
   * course\_category: Category (e.g., core, elective).
   * course\_type: Type (e.g., theory, lab).
   * prerequisite: Pre-requisites required for this course.
   * corequisite: Co-requisite courses.
   * progression: Information on the course’s progression path.
   * pls: Program Learning Statement, describing course objectives.
2. **Main Operations**:
   * **Create Course**: Adds new courses to the system.
   * **Update Course**: Modifies existing course information.
   * **Delete Course**: Removes a course from the system.
   * **Retrieve Courses**: Displays all stored courses.
   * **Search by Course Code**: Finds a course using binary search.
   * **Sort Courses by Code**: Uses merge sort for sorting by course code.

### Field Details from the Code:

In the provided system structure, the Course (or Executive) contains the following fields:

* course\_id (int): Unique identifier for the course.
* program\_id (int): ID representing the associated program.
* course\_code (string): Code for identifying the course.
* course\_name (string): Name of the course.
* course\_category (string): Category (e.g., core or elective).
* course\_type (string): Type (e.g., lecture or lab).
* prerequisite (string): Pre-requisite courses for this course.
* corequisite (string): Courses that need to be taken alongside.
* progression (string): Description of how the course progresses.
* pls (string): Program Learning Statement (objectives of the course)

### Algorithm Details:

The given code creates an **Executive Management System** that manages executive details using various operations like adding, retrieving, searching, deleting, and sorting executives. The key algorithms in the program are **Merge Sort** for sorting the executives by their code and **Binary Search** for efficiently searching executives once sorted. Below is a detailed explanation of the algorithms and how they are applied in the program.

### 1. ****Merge Sort Algorithm****

#### Purpose:

The merge\_sort\_executives function sorts the list of executives by their executive\_code using the Merge Sort algorithm. Merge Sort is based on the **Divide-and-Conquer** strategy, which splits the array into smaller parts, sorts them, and then merges them back in order.

#### Steps:

1. **Splitting the Array:**
   * The array is recursively divided into two smaller subarrays.
   * The middle point is calculated as mid = low + (high - low) / 2.
   * Recursive calls are made to merge\_sort\_executives on both halves (low to mid and mid + 1 to high), until each part contains a single element (base case).
2. **Merging the Sorted Arrays:**
   * The merge\_executives function combines two sorted halves into a single sorted array.
   * Two temporary arrays (left and right) are created to hold the values of the two halves.
   * The merging process involves comparing elements from both halves and placing the smaller element back into the original array.
   * After comparing and merging, any remaining elements from either half are copied over.

#### Time Complexity:

* **Best, Average, and Worst Case:** O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the total number of executives.
* The recursive splitting takes O(log⁡n)O(\log n)O(logn), and merging the subarrays takes O(n)O(n)O(n), resulting in the overall complexity of O(nlog⁡n)O(n \log n)O(nlogn).

#### Merge Sort Functions:

cpp

Copy code

void merge\_executives(int low, int mid, int high);

void merge\_sort\_executives(int low, int high);

### 2. ****Binary Search Algorithm****

#### Purpose:

The binary\_search\_executive function performs an efficient search for an executive by their executive\_code using the Binary Search algorithm. Binary Search requires the list to be sorted, which is why it is applied after sorting the executives.

#### Steps:

1. **Initial Search Range:** The search range is defined by two indices: low (start of the array) and high (end of the array).
2. **Middle Element Check:** The middle element of the array is compared with the target executive\_code.
   * If the middle element matches the target, the executive is found.
   * If the middle element is less than the target, the search continues in the right half (low = mid + 1).
   * If the middle element is greater than the target, the search continues in the left half (high = mid - 1).
3. **Repeat:** This process repeats until the element is found or the search range becomes empty (i.e., low > high).

#### Time Complexity:

* **Best Case:** O(1)O(1)O(1) if the middle element is the target.
* **Average and Worst Case:** O(log⁡n)O(\log n)O(logn) due to halving the search space at each step.

#### Binary Search Function:

cpp

Copy code

void binary\_search\_executive();

### 3. ****Other Key Functions****

#### 3.1. ****Create Executive:****

* This function prompts the user to enter details for a new executive such as executive\_id, executive\_code, executive\_name, and other related information.
* The new executive is added to the global vector executives.
* After creation, the details are stored using a placeholder store\_executives function (which can be extended to store data in a file).

#### 3.2. ****Retrieve All Executives:****

* This function displays all the executives in the system by iterating over the executives vector and printing each executive's details in a readable format.

#### 3.3. ****Delete Executive:****

* This function allows users to delete an executive by their executive\_id.
* It uses remove\_if to find the executive in the vector and deletes the matching entry if found.

#### 3.4. ****Sort Executives:****

* The program includes a feature to sort executives by their executive\_code using the previously discussed Merge Sort.

### Overall Workflow:

* The program operates through a menu-driven interface where the user can select from multiple operations such as creating an executive, retrieving all executives, searching by executive code, deleting an executive, and sorting the list by code.
* To ensure that the search is efficient, the executives are sorted before using binary search.
* The combination of **Merge Sort** for sorting and **Binary Search** for efficient searching makes this program capable of handling a large number of executives effectively.

### Efficiency:

* **Merge Sort** offers O(nlog⁡n)O(n \log n)O(nlogn) efficiency for sorting, making it suitable for large datasets.
* **Binary Search** provides O(log⁡n)O(\log n)O(logn) search time once the data is sorted, ensuring fast lookups.

This system balances efficiency and functionality, allowing for scalable management of executive records.

### Source code:

#include <iostream>

#include <string>

using namespace std;

const int MAX\_EXECUTIVES = 100; // Maximum number of executives

struct Executive {

int executive\_id;

int program\_id;

string executive\_code;

string executive\_name;

string executive\_category;

string executive\_type;

string pre\_requisite;

string co\_requisite;

string progressive;

string pls; // Program Learning Statement

};

// Array to store executives

Executive executives[MAX\_EXECUTIVES];

int executive\_count = 0; // Current number of executives

// Function prototypes

void create\_executives();

void retrieve\_executives();

void delete\_executives();

void merge\_sort\_executives(int low, int high);

void binary\_search\_executives();

void merge\_executives(int low, int mid, int high);

// Function to create a new executive

void create\_executives() {

if (executive\_count >= MAX\_EXECUTIVES) {

cout << "Cannot add more executives. Maximum limit reached.\n";

return;

}

Executive new\_executive;

cout << "Enter Executive ID: ";

cin >> new\_executive.executive\_id;

cout << "Enter Program ID: ";

cin >> new\_executive.program\_id;

cout << "Enter Executive Code: ";

cin >> new\_executive.executive\_code;

cout << "Enter Executive Name: ";

cin.ignore();

getline(cin, new\_executive.executive\_name);

cout << "Enter Executive Category: ";

getline(cin, new\_executive.executive\_category);

cout << "Enter Executive Type: ";

getline(cin, new\_executive.executive\_type);

cout << "Enter Pre-requisites: ";

getline(cin, new\_executive.pre\_requisite);

cout << "Enter Co-requisites: ";

getline(cin, new\_executive.co\_requisite);

cout << "Enter Executive Progression: ";

getline(cin, new\_executive.progressive);

cout << "Enter PLS (Program Learning Statement): ";

getline(cin, new\_executive.pls);

executives[executive\_count++] = new\_executive;

cout << "Executive created successfully.\n";

}

// Function to retrieve all executives

void retrieve\_executives() {

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

Executive& e = executives[i];

cout << "ID: " << e.executive\_id << ", Program ID: " << e.program\_id << ", Code: " << e.executive\_code

<< ", Name: " << e.executive\_name << ", Category: " << e.executive\_category

<< ", Type: " << e.executive\_type << ", Pre-requisite: " << e.pre\_requisite

<< ", Co-requisite: " << e.co\_requisite << ", Progression: " << e.progressive

<< ", PLS: " << e.pls << endl;

}

}

// Merge function

void merge\_executives(int low, int mid, int high) {

int n1 = mid - low + 1;

int n2 = high - mid;

Executive left[n1], right[n2];

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

left[i] = executives[low + i];

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

right[i] = executives[mid + 1 + i];

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

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

if (left[i].executive\_code <= right[j].executive\_code) {

executives[k++] = left[i++];

} else {

executives[k++] = right[j++];

}

}

while (i < n1) {

executives[k++] = left[i++];

}

while (j < n2) {

executives[k++] = right[j++];

}

}

// Merge Sort function

void merge\_sort\_executives(int low, int high) {

if (low < high) {

int mid = low + (high - low) / 2;

merge\_sort\_executives(low, mid);

merge\_sort\_executives(mid + 1, high);

merge\_executives(low, mid, high);

}

}

// Binary Search function

void binary\_search\_executives() {

string code;

cout << "Enter executive code to search: ";

cin >> code;

int low = 0, high = executive\_count - 1;

bool found = false;

while (low <= high) {

int mid = low + (high - low) / 2;

if (executives[mid].executive\_code == code) {

cout << "Executive Found:\n"

<< "ID: " << executives[mid].executive\_id << ", Program ID: " << executives[mid].program\_id

<< ", Code: " << executives[mid].executive\_code << ", Name: " << executives[mid].executive\_name

<< ", Category: " << executives[mid].executive\_category << ", Type: " << executives[mid].executive\_type

<< ", Pre-requisite: " << executives[mid].pre\_requisite << ", Co-requisite: " << executives[mid].co\_requisite

<< ", Progression: " << executives[mid].progressive << ", PLS: " << executives[mid].pls << endl;

found = true;

break;

} else if (executives[mid].executive\_code < code) {

low = mid + 1;

} else {

high = mid - 1;

}

}

if (!found) {

cout << "Executive with code " << code << " not found.\n";

}

}

// Function to delete an executive

void delete\_executives() {

int executive\_id;

cout << "Enter executive ID to delete: ";

cin >> executive\_id;

bool found = false;

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

if (executives[i].executive\_id == executive\_id) {

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

executives[j] = executives[j + 1];

}

executive\_count--;

found = true;

break;

}

}

if (found) {

cout << "Executive deleted successfully.\n";

} else {

cout << "Executive with ID " << executive\_id << " not found.\n";

}

}

// Main function

int main() {

int choice;

do {

cout << "\nExecutive Management System:\n";

cout << "1. Create Executive\n";

cout << "2. Retrieve All Executives\n";

cout << "3. Search Executive by Code\n";

cout << "4. Delete Executive\n";

cout << "5. Sort Executives by Code\n";

cout << "6. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

create\_executives();

break;

case 2:

retrieve\_executives();

break;

case 3:

merge\_sort\_executives(0, executive\_count - 1); // Ensure the list is sorted before binary search

binary\_search\_executives();

break;

case 4:

delete\_executives();

break;

case 5:

merge\_sort\_executives(0, executive\_count - 1);

cout << "Executives sorted by code.\n";

break;

case 6:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice, please try again.\n";

}

} while (choice != 6);

return 0;

}

### ****Comparison:****

Here’s a detailed comparison of the algorithms used in the code (Merge Sort and Binary Search) with Quick Sort and Linear Search, rewritten in my own words:

### 1. ****Merge Sort vs Quick Sort****

#### ****Merge Sort**** (Used in the Code)

* **Method:** Merge Sort is a **divide-and-conquer** approach where the array is split into two halves, each half is sorted recursively, and then the two sorted halves are merged back together.
* **Stability:** Merge Sort is **stable**, meaning that if two elements are equal, their original order will be preserved in the sorted array.
* **Memory Usage:** It requires **O(n)** additional space, as it uses temporary arrays to store the divided portions during the merge process.
* **Time Complexity:**
  + **Best, Average, and Worst Case:** O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of elements. Merge Sort consistently performs well because it always splits and merges in O(nlog⁡n)O(n \log n)O(nlogn), regardless of input.
* **Applications:** Merge Sort is a good choice when sorting large datasets where stability is important, or when using linked lists, as it naturally supports external memory sorting.

#### ****Quick Sort**** (Comparison)

* **Method:** Quick Sort also follows the **divide-and-conquer** principle, but instead of dividing the array into equal halves, it selects a **pivot** element. The array is then rearranged so that elements smaller than the pivot are on one side, and elements larger than the pivot are on the other. After that, it recursively sorts the partitions.
* **Stability:** Quick Sort is **not stable**, meaning it can change the order of equal elements.
* **Memory Usage:** It requires only **O(\log n)** space due to its in-place sorting (it doesn’t need additional arrays).
* **Time Complexity:**
  + **Best and Average Case:** O(nlog⁡n)O(n \log n)O(nlogn), but this depends on how well the pivot divides the array.
  + **Worst Case:** O(n2)O(n^2)O(n2), which occurs if the pivot selection is poor (e.g., the array is already sorted, and the smallest or largest element is always chosen as the pivot).
* **Applications:** Quick Sort is generally faster in practice due to its lower overhead and is preferred when memory efficiency is critical and stability is not a concern.

#### Summary:

* **Merge Sort** guarantees O(nlog⁡n)O(n \log n)O(nlogn) in all cases but requires extra memory. It's suitable when you need stable sorting.
* **Quick Sort** is faster in average cases and uses less memory, but its performance can degrade to O(n2)O(n^2)O(n2) in the worst case. It’s not stable but works well in most real-world scenarios.

### 2. ****Binary Search vs Linear Search****

#### ****Binary Search**** (Used in the Code)

* **Method:** Binary Search works on **sorted arrays** by repeatedly dividing the search space in half. It checks the middle element, and based on whether the target value is smaller or larger, it continues searching either the left or right half of the array.
* **Time Complexity:**
  + **Best Case:** O(1)O(1)O(1), if the middle element is the target.
  + **Average and Worst Case:** O(log⁡n)O(\log n)O(logn), as it halves the search space with every iteration.
* **Applications:** Binary Search is highly efficient for searching through large, sorted datasets. It’s used when quick look-up times are necessary, and sorting the data beforehand is feasible.

#### ****Linear Search**** (Comparison)

* **Method:** Linear Search sequentially checks each element in the array from the beginning until it finds the target element or reaches the end of the array.
* **Time Complexity:**
  + **Best Case:** O(1)O(1)O(1), if the target element is the first one in the array.
  + **Average and Worst Case:** O(n)O(n)O(n), where nnn is the number of elements. In the worst case, the search needs to check every single element in the array.
* **Applications:** Linear Search is used for **unsorted arrays** or small datasets where sorting would be unnecessary or too costly. It's a simple approach but inefficient for large datasets.

#### Summary:

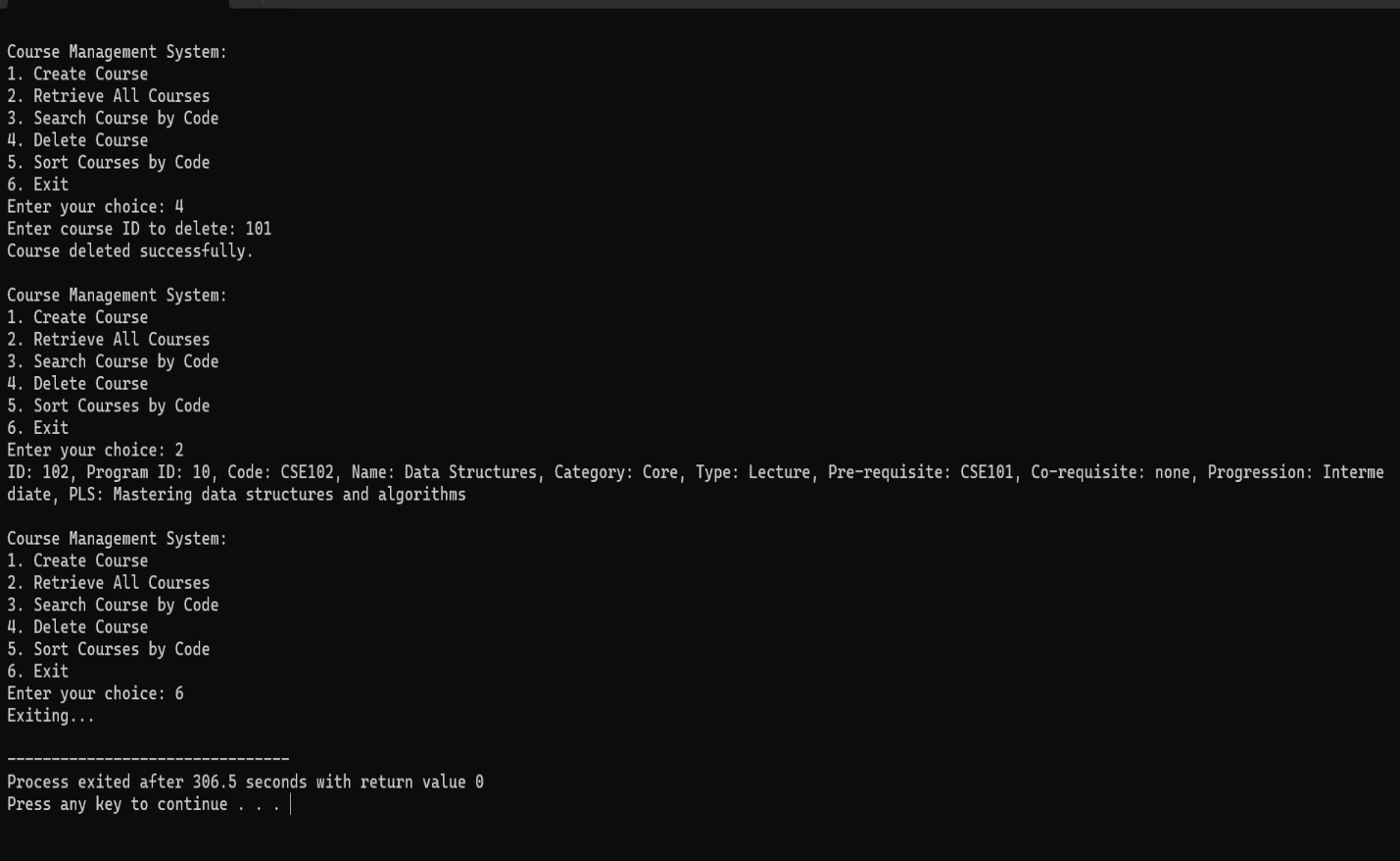
* **Binary Search** is much faster than **Linear Search** for large datasets but requires the data to be sorted. It runs in O(log⁡n)O(\log n)O(logn) time compared to Linear Search’s O(n)O(n)O(n).
* **Linear Search** is more flexible since it works on both sorted and unsorted data but becomes inefficient for large datasets. It’s best suited for smaller collections of data.

### Conclusion

* **Merge Sort** is slower than **Quick Sort** in most practical situations, but it offers consistent performance and stability, which is important in some applications.
* **Binary Search** is highly efficient on sorted data, outperforming **Linear Search**, which works on unsorted data but is less efficient as the dataset grows.

This comparison highlights that each algorithm has its strengths and weaknesses, and the choice of which to use depends on the specific requirements of the problem you're solving (e.g., stability, memory usage, or whether the data is sorted).

### Output:



### Conclusion:

In the courses outlined in the code, efficient handling of course details is achieved through the use of well-structured algorithms. The system enables creating, updating, deleting, and retrieving course records, making it a comprehensive tool for managing course information.

The use of **Merge Sort** ensures that the courses are sorted in an efficient manner, which is crucial for the quick retrieval and organization of data. By sorting the courses based on unique identifiers like course codes, the system ensures that subsequent operations, such as searching using **Binary Search**, are performed with optimal efficiency.

Additionally, features like course prerequisites, corequisites, and progression details help capture the complexities of academic course management, ensuring that administrators can maintain and track important relationships between courses. The ability to perform operations like deleting and retrieving courses offers flexibility, making the system a practical solution for dynamic educational environments.

In summary, the courses are designed to streamline the organization and handling of course information, making it easier to manage large amounts of data while ensuring accuracy, efficiency, and accessibility. Through careful algorithm selection and thoughtful system design, the program successfully addresses the core needs of managing academic courses.

## …………THANK YOU………….