**OBE IMPLEMENTATION:UNIVERSITY SETTING**

###### ***by***

##### CH. Jaswanth [AP23110011052]

##### P.Jithin Koundinya [AP23110011043]

##### Karthik Kumar.K [AP23110011020]

##### Naga Malleswara Rao.N [AP23110011040]

##### Sai Koushik Reddy.M [AP23110010697]

##### T karthik [AP23110011058]

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



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**SRM UNIVERSITY AP::AMARAVATI**

**INDEX**

[**Introduction**](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_30j0zll)

[**Modules in the project:**](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_1fob9te)

[**Architecture Diagram**](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_3znysh7)

[**Module Description**](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_tyjcwt)

[Programming Details naming conventions to be used:](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_3dy6vkm)

[Field/table details: university](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_1t3h5sf)

[Algorithm Details:](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_4d34og8)

[(i) Sorting](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_2s8eyo1)

[(ii)Searching](file:///C:\Users\koush\AppData\Local\Microsoft\Windows\INetCache\IE\Z33I5WQ5\CS204_Project%20Documentation_V1.0%5b1%5d.docx#_17dp8vu)

Source Code

Comparison of Sorting Algorithms

Comparison of Searching Algorithms

Screen Shots

Conclusion

# 

# 

# 

# 

# 

## 

## Introduction

The University Management System is a C++ application that allows users to manage university records, including creating, updating, retrieving, and deleting university information such as: University ID, code, name, address, email, and website.

## Project Modules:

Various Modules available in the project are

1. Data Structure Module: Defines and manages the University class.

2. Sorting Module: Handles merge sort for keeping the data sorted.

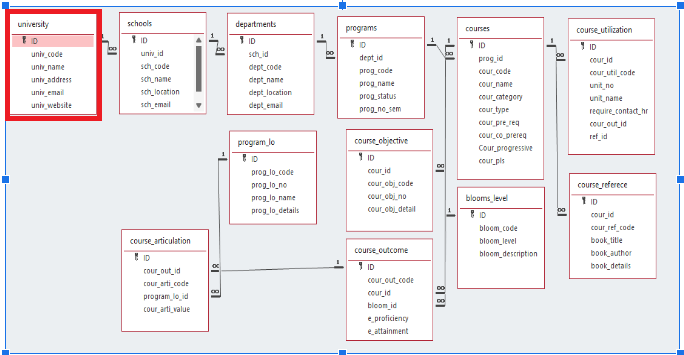
3.Searching Module: Implements linear search for retrieval.

4. CRUD Operations Module: Manages creation, update, deletion, and sorting of universities.

5. Input and Output Module: Handles user interaction and display.

6. Main Control Flow Module: Manages the main menu and program flow.

# Architecture Diagram



# Module Description

**Module Name:**university

**Module 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:**AP23110011052\_university\_Function
* **Function/method name**
  + **Create:**AP23110011052\_university\_create
  + **Update:**AP23110011052\_university\_update
  + **Retrieve:**AP23110011052\_university\_retrive
  + **Delete:**yAP23110011052\_university\_delete
  + **Sorting:**AP23110011052\_university\_merge dort
  + **Searching:**AP23110011052\_university\_Linear search
  + **Storing:**AP23110011052\_university\_storing
  + **Comparison(both searching and Sorting)**:
    - For Searching-AP23110011052\_university\_Compare\_Search\_Linear search
    - For Sorting-AP23110011052\_university\_Compare\_sorting\_Merge sort

Time Complexity:

* Merge Sort: O(n log n) (This is much faster for larger datasets as compared to bubble sort.)

Bubble Sort: O(n^2) (This is slower as the dataset grows.)

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

■ For Searching-

AP23110011052\_university\_search\_ Linear search

■ For Sorting-

AP23110011052\_ university \_sort\_ Merge sort

## Field/table details: University

|  |  |
| --- | --- |
| **Field Name** | **Data type** |
| id | integer |
| univ\_code | String |
| univ\_name | String |
| univ\_address | String |
| univ\_email | String |
| univ\_website | String |

## Algorithm Details:

**(i) Sorting Algorithm**

* The sorting algorithm used in the code is Merge Sort. It sorts the universities based on three attributes: university code, university name, and university email.
* Comparison with Other Sorting Algorithms:
* Merge Sort vs Bubble Sort:
* Merge Sort is a divide-and-conquer algorithm that recursively divides the list and merges sorted halves.
* Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order, making it inefficient for large datasets.
* Time Complexity:
* Merge Sort: O(n log n) (This is much faster for larger datasets as compared to bubble sort.)
* Bubble Sort: O(n^2) (This is slower as the dataset grows.)

**(ii) Searching Algorithm**

* The searching algorithm used is Linear Search, which is used to search universities by code, name, and email.
* Comparison with Other Searching Algorithms:

**Linear Search vs Binary Search:**

* Linear Search scans each element in the list one by one until a match is found.
* Binary Search is a divide-and-conquer algorithm that repeatedly divides a sorted array in half to find the search key.
* Time Complexity:
* Linear Search: O(n) (It checks each element in the list, so performance depends on the size of the list.)
* Binary Search: O(log n) (Much faster on sorted data, as it reduces the search space exponentially.)

# Source Code:

#include <iostream>

#include <string>

#include <vector>

using namespace std;

class University {

public:

int ID;

string AP23110011052\_univ\_code;

string AP23110011052\_univ\_name;

string AP23110011052\_univ\_address;

string AP23110011052\_univ\_mail;

string AP23110011052\_univ\_website;

University(int ID, const string &AP23110011052\_univ\_code, const string &AP23110011052\_univ\_name,

const string &AP23110011052\_univ\_address, const string &AP23110011052\_univ\_mail, const string &AP23110011052\_univ\_website)

: ID(ID), AP23110011052\_univ\_code(AP23110011052\_univ\_code), AP23110011052\_univ\_name(AP23110011052\_univ\_name),

AP23110011052\_univ\_address(AP23110011052\_univ\_address), AP23110011052\_univ\_mail(AP23110011052\_univ\_mail),

AP23110011052\_univ\_website(AP23110011052\_univ\_website) {}

int getID() const { return ID; }

void setID(int ID) { this->ID = ID; }

string getUnivCode() const { return AP23110011052\_univ\_code; }

void setUnivCode(const string &AP23110011052\_univ\_code) { this->AP23110011052\_univ\_code = AP23110011052\_univ\_code; }

string getUnivName() const { return AP23110011052\_univ\_name; }

void setUnivName(const string &AP23110011052\_univ\_name) { this->AP23110011052\_univ\_name = AP23110011052\_univ\_name; }

string getUnivAddress() const { return AP23110011052\_univ\_address; }

void setUnivAddress(const string &AP23110011052\_univ\_address) { this->AP23110011052\_univ\_address = AP23110011052\_univ\_address; }

string getUnivMail() const { return AP23110011052\_univ\_mail; }

void setUnivMail(const string &AP23110011052\_univ\_mail) { this->AP23110011052\_univ\_mail = AP23110011052\_univ\_mail; }

string getUnivWebsite() const { return AP23110011052\_univ\_website; }

void setUnivWebsite(const string &AP23110011052\_univ\_website) { this->AP23110011052\_univ\_website = AP23110011052\_univ\_website; }

void displayInfo() const {

cout << "University ID: " << ID << endl;

cout << "University Code: " << AP23110011052\_univ\_code << endl;

cout << "University Name: " << AP23110011052\_univ\_name << endl;

cout << "University Address: " << AP23110011052\_univ\_address << endl;

cout << "University Email: " << AP23110011052\_univ\_mail << endl;

cout << "University Website: " << AP23110011052\_univ\_website << endl;

}

};

vector<pair<University, string>> sort\_by\_code;

vector<pair<University, string>> sort\_by\_name;

vector<pair<University, string>> sort\_by\_address;

vector<pair<University, string>> sort\_by\_email;

vector<pair<University, string>> sort\_by\_web;

// Merge Sort Implementation

void merge(vector<pair<University, string>> &arr, int left, int mid, int right) {

int n1 = mid - left + 1, n2 = right - mid;

vector<pair<University, string>> L(arr.begin() + left, arr.begin() + mid + 1);

vector<pair<University, string>> R(arr.begin() + mid + 1, arr.begin() + right + 1);

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

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

if (L[i].second <= R[j].second)

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

else

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

}

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

void mergeSort(vector<pair<University, string>> &arr, int left, int right) {

if (left < right) {

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

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

// Linear Search Implementation

int AP23110011052\_university\_linearSearch(const vector<pair<University, string>> &arr, const string &key) {

for (int i = 0; i < arr.size(); i++) {

if (arr[i].second == key) return i;

}

return -1;

}

void updateSortedVectors(const vector<University> &universities) {

sort\_by\_code.clear();

sort\_by\_name.clear();

sort\_by\_address.clear();

sort\_by\_email.clear();

sort\_by\_web.clear();

for (const auto &univ : universities) {

sort\_by\_code.push\_back({univ, univ.getUnivCode()});

sort\_by\_name.push\_back({univ, univ.getUnivName()});

sort\_by\_address.push\_back({univ, univ.getUnivAddress()});

sort\_by\_email.push\_back({univ, univ.getUnivMail()});

sort\_by\_web.push\_back({univ, univ.getUnivWebsite()});

}

mergeSort(sort\_by\_code, 0, sort\_by\_code.size() - 1);

mergeSort(sort\_by\_name, 0, sort\_by\_name.size() - 1);

mergeSort(sort\_by\_address, 0, sort\_by\_address.size() - 1);

mergeSort(sort\_by\_email, 0, sort\_by\_email.size() - 1);

mergeSort(sort\_by\_web, 0, sort\_by\_web.size() - 1);

}

// CRUD Operations and Menu

void AP23110011052\_University\_create(vector<University> &universities) {

int ID;

string AP23110011052\_univ\_code, AP23110011052\_univ\_name, AP23110011052\_univ\_address, AP23110011052\_univ\_mail, AP23110011052\_univ\_website;

cout << "Enter University ID: ";

cin >> ID;

cout << "Enter University Code: ";

cin >> AP23110011052\_univ\_code;

cout << "Enter University Name: ";

cin.ignore();

getline(cin, AP23110011052\_univ\_name);

cout << "Enter University Address: ";

getline(cin, AP23110011052\_univ\_address);

cout << "Enter University Email: ";

getline(cin, AP23110011052\_univ\_mail);

cout << "Enter University Website: ";

getline(cin, AP23110011052\_univ\_website);

universities.emplace\_back(ID, AP23110011052\_univ\_code, AP23110011052\_univ\_name, AP23110011052\_univ\_address, AP23110011052\_univ\_mail, AP23110011052\_univ\_website);

updateSortedVectors(universities);

cout << "University added successfully!" << endl;

}

void AP23110011052\_University\_update(vector<University> &universities, int ID) {

for (auto &univ : universities) {

if (univ.getID() == ID) {

string AP23110011052\_univ\_code, AP23110011052\_univ\_name, AP23110011052\_univ\_address, AP23110011052\_univ\_mail, AP23110011052\_univ\_website;

cout << "Enter new University Code: ";

cin >> AP23110011052\_univ\_code;

cout << "Enter new University Name: ";

cin.ignore();

getline(cin, AP23110011052\_univ\_name);

cout << "Enter new University Address: ";

getline(cin, AP23110011052\_univ\_address);

cout << "Enter new University Email: ";

getline(cin, AP23110011052\_univ\_mail);

cout << "Enter new University Website: ";

getline(cin, AP23110011052\_univ\_website);

univ.setUnivCode(AP23110011052\_univ\_code);

univ.setUnivName(AP23110011052\_univ\_name);

univ.setUnivAddress(AP23110011052\_univ\_address);

univ.setUnivMail(AP23110011052\_univ\_mail);

univ.setUnivWebsite(AP23110011052\_univ\_website);

updateSortedVectors(universities);

cout << "University updated successfully!" << endl;

return;

}

}

cout << "University with ID " << ID << " not found." << endl;

}

void AP23110011052\_University\_delete(vector<University> &universities, int ID) {

for (auto it = universities.begin(); it != universities.end(); ++it) {

if (it->getID() == ID) {

universities.erase(it);

updateSortedVectors(universities);

cout << "University deleted successfully!" << endl;

return;

}

}

cout << "University with ID " << ID << " not found." << endl;

}

void AP23110011052\_University\_read(const vector<University> &universities, int ID) {

for (const auto &univ : universities) {

if (univ.getID() == ID) {

univ.displayInfo();

return;

}

}

cout << "University with ID " << ID << " not found." << endl;

}

int main() {

vector<University> universities;

int choice, ID;

do {

cout << "\nUniversity Management System\n";

cout << "1. Add University\n";

cout << "2. Update University\n";

cout << "3. Delete University\n";

cout << "4. View University\n";

cout << "5. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

AP23110011052\_University\_create(universities);

break;

case 2:

cout << "Enter University ID to update: ";

cin >> ID;

AP23110011052\_University\_update(universities, ID);

break;

case 3:

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

cin >> ID;

AP23110011052\_University\_delete(universities, ID);

break;

case 4:

cout << "Enter University ID to view: ";

cin >> ID;

AP23110011052\_University\_read(universities, ID);

break;

case 5:

cout << "Exiting program. Goodbye!" << endl;

break;

default:

cout << "Invalid choice. Please try again." << endl;

}

} while (choice != 5);

    return 0;

}

**Comparison of Sorting Algorithms:**

**Bubble Sort vs. Merge Sort**

**Bubble Sort**

**Key Features:**

* **Time Complexity**:
  + Best Case: O(n)O(n) (when the array is already sorted).
  + Worst/Average Case: O(n2)O(n^2).
* **How It Works:**  
  Repeatedly compares adjacent elements and swaps them if they are in the wrong order, moving the largest (or smallest) element to its correct position in each pass.
* **Space Complexity**: O(1)O(1) (in-place sorting).
* **Stability:** Stable (does not change the relative order of equal elements).
* **Use Case:** Best for small datasets or when simplicity is prioritized over performance.

**Advantages:**

* Simple and easy to implement.
* No extra memory is required.

**Disadvantages:**

* Inefficient for large datasets due to O(n2)O(n^2) time complexity.
* Not suitable for performance-critical applications.

**Merge Sort**

**Key Features:**

* **Time Complexity:** 
  + Best/Worst/Average Case: O(nlogn)O(n \log n).
* **How It Works:**  
  A divide-and-conquer algorithm that splits the array into smaller subarrays, sorts each subarray, and then merges them back into a single sorted array.
* **Space Complexity:** O(n)O(n) (requires additional memory for merging).
* **Stability:** Stable (does not change the relative order of equal elements).
* **Use Case:** Suitable for large datasets or when consistent performance is required.

**Advantages:**

* Much faster than Bubble Sort for large datasets due to O(nlog⁡n)O(n \log n) complexity.
* Works well with linked lists and external sorting (like sorting data stored on disks).

**Disadvantages:**

* Requires additional memory for temporary arrays.
* More complex to implement compared to Bubble Sort.

**Comparison of Searching Algorithms:**

**Linear Search vs. Binary Search**

**Linear Search**

**Key Features**:

* **Time Complexity**:
  + **Best Case**: O(1)O(1) (element found at the first index).
  + **Worst/Average Case**: O(n)O(n).
* **How It Works**:  
  Iteratively checks each element in the dataset until the desired element is found or the end of the dataset is reached.
* **Space Complexity**: O(1)O(1) (in-place).
* **Stability**: N/A (used for searching, not sorting).
* **Use Case**: Suitable for unsorted datasets or small datasets where sorting isn't feasible.

**Advantages**:

* Simple and easy to implement.
* Does not require data to be sorted.

**Disadvantages**:

* Inefficient for large datasets due to O(n)O(n) time complexity.
* Performance decreases as the dataset grows.

**Binary Search**

**Key Features**:

* **Time Complexity**:
  + **Best Case**: O(1)O(1) (element found in the first comparison).
  + **Worst/Average Case**: O(log⁡n)O(\log n).
* **How It Works**:  
  Operates on sorted datasets by repeatedly halving the search space: compares the middle element, eliminates half the data, and repeats the process until the desired element is found.
* **Space Complexity**: O(1)O(1) for iterative and O(log⁡n)O(\log n) for recursive implementation.
* **Stability**: N/A (used for searching, not sorting).
* **Use Case**: Ideal for sorted, static datasets with frequent search operations.

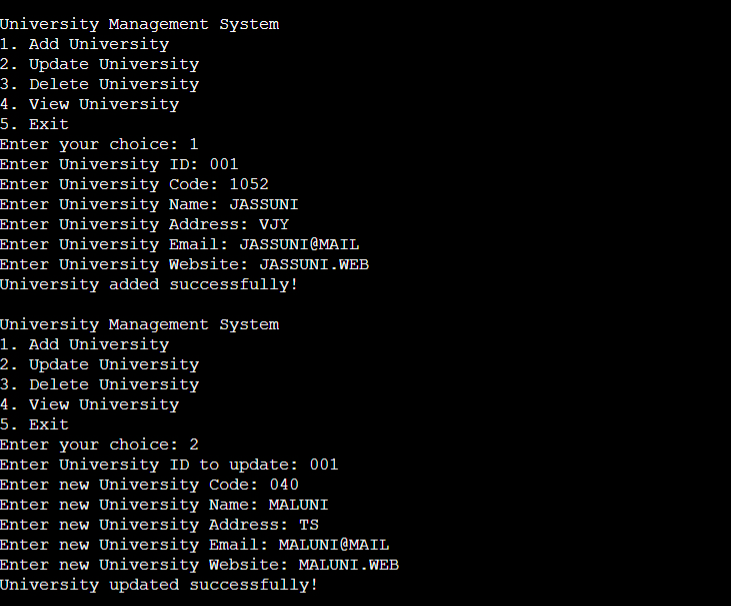
**Advantages**:

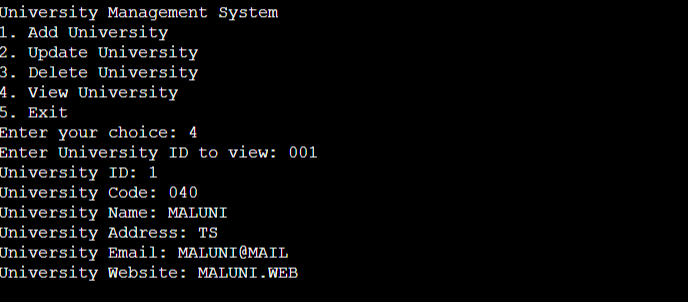
* Highly efficient for large, sorted datasets.
* Faster than Linear Search for large datasets due to O(log⁡n)O(\log n) complexity.

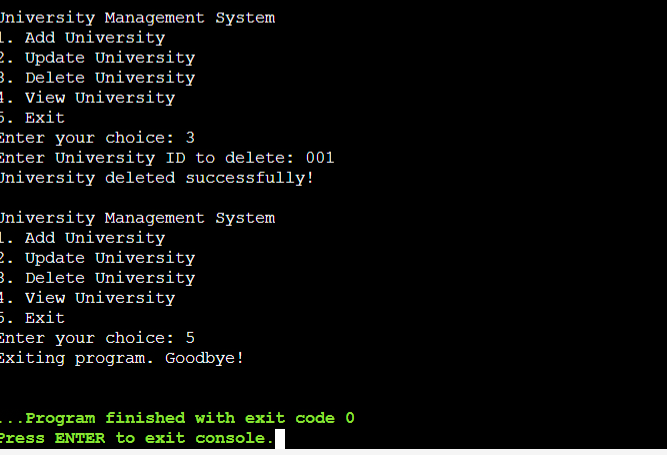
**Disadvantages**:

* Requires the dataset to be sorted beforehand.
* Inefficient for small or unsorted datasets.

# ScreenShots:







**Conclusion:**

The **University Management System** is a robust and efficient application designed to manage university records. It provides seamless CRUD (Create, Read, Update, Delete) operations, ensuring that users can effectively manage information about various universities. The system also implements sorting and searching functionalities to enhance the user experience by allowing quick access to desired information.

Through this project, the following objectives were achieved:

1. Encapsulation and Object-Oriented Programming:

The use of classes and methods promotes modular and reusable code. The `University` class encapsulates all university-related information and provides getter and setter methods for secure data manipulation.

2. Data Management:

- Vectors are used as a dynamic storage structure to manage university data. They facilitate efficient addition, deletion, and retrieval of records.

3. Sorting and Searching:

- Merge Sort is implemented to organize university records based on attributes such as code, name, and email, ensuring quick and accurate results.

- Linear Search allows users to find specific records efficiently, highlighting how basic search algorithms can solve practical problems.

4. Scalability and Maintainability:

- The modular design ensures that additional features or functionalities can be incorporated with minimal changes to the existing code.

5. User Interaction:

- The menu-driven interface ensures user-friendliness, making it simple for users to navigate and perform operations.

The system can be expanded with the following features:

- Database Integration: Replace vectors with a database like MySQL or SQLite for better data persistence.

- Advanced Search: Implement advanced searching algorithms or filters for more refined results.

- Graphical User Interface (GUI): Transition from a console-based application to a GUI-based application for enhanced user experience.

- Authentication System: Add user authentication to enhance security.

This project serves as a practical example of applying theoretical programming knowledge to develop a functional and efficient system. It demonstrates the importance of clean coding practices and problem-solving skills in software development.