SRM UNIVERSITY AP
Amaravati

OBE Implementation

Module-1:COURSE

Submitted By:

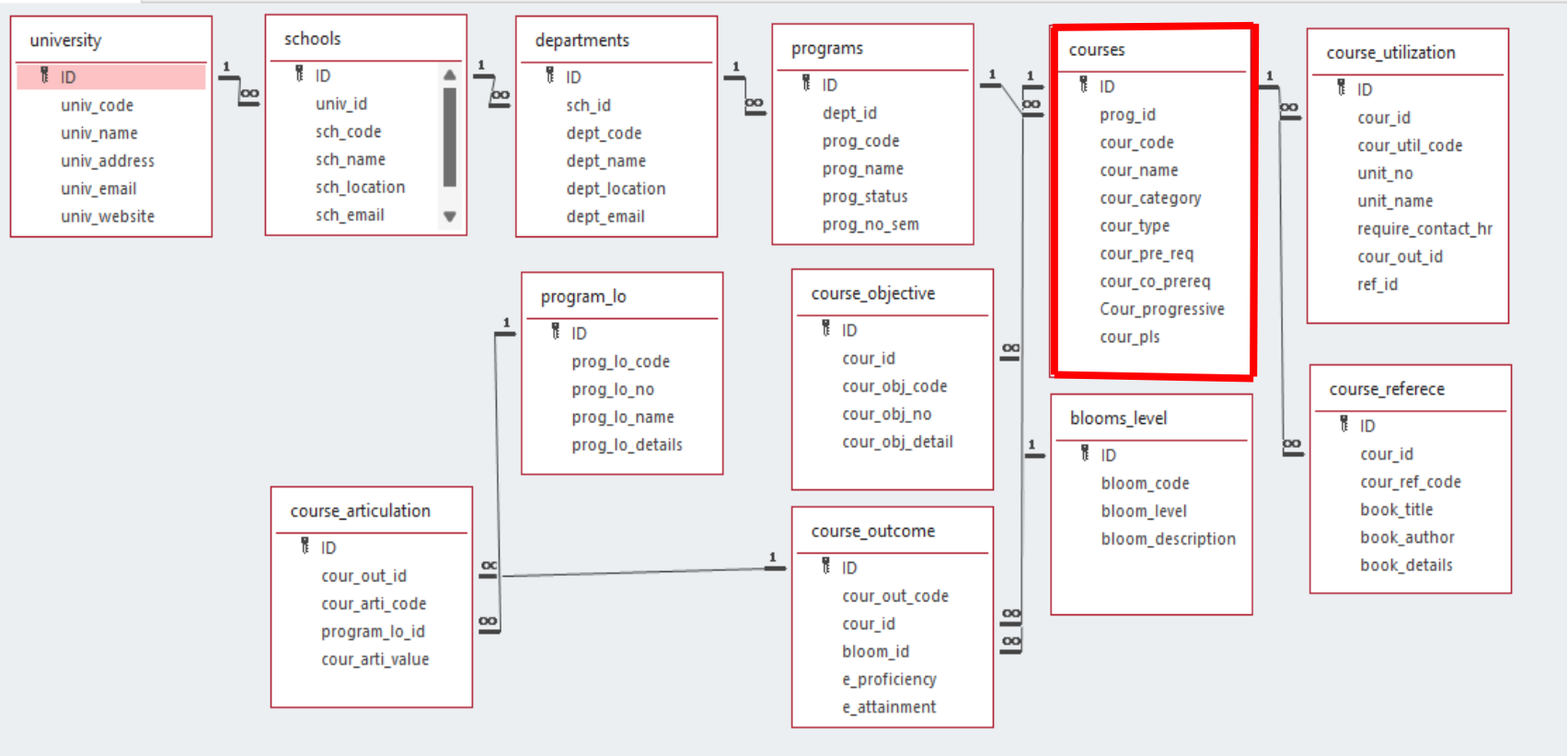
S. Rupali - AP23110011081
P. Chashmitha - AP23110011087
G. Geethika - AP23110011104
T. Varshitha - AP23110011136
K. Harshavardhini - AP23110011144
Sk. Farzana - AP23110011149

CSE | Sem - 3 | B.Tech

Introduction to Project

The course will introduce students to the fundamentals of **sorting** and **searching algorithms** while emphasizing their **outcome-based implementation**. The goal is for students to not only understand the theory behind algorithms but also to implement them, analyze their time complexity, and compare them in terms of efficiency and performance.

Architecture Diagram



Module Description : COURSE Setting

- Sorting and searching are two fundamental algorithmic operations that play a key role in computer science and software development. Understanding these algorithms is crucial for solving a wide range of problems efficiently. Below is a summary of **sorting** and **searching algorithms**, focusing on their core concepts, differences, and key takeaways.

COURSE Setting:Field/table details

FIELD NAME	DATA TYPE
id	Integer
prog_id	String
cour_code	String
cour_name	String
cour_category	String
cour_type	String
cour_pre_req	String
cour_co_prereq	String
cour_progressive	String
cour_pls	String

Course Setting:Programming Details

File name: Courses
Function/method name
Create:Courses_create_program
Update:Courses_update_program
Retrieve:Courses_retrive_program
Delete:Courses_delete_program
Sorting: Courses_sort_by_field
Searching:courses_search_by_field

Course Setting:Programming Details

- **Comparison(both searching and Sorting)**:
  - For Searching:-Courses_Compare_Searching_algorithms
  - For Sorting:-Courses_Compare_sorting_your algorithm
  - **Time Complexity(both searching and Sorting):**
  - For Searching:-Courses_complexity_Search
  - For Sorting:-Courses_compexity_sorting

Course Setting:Programming Details

- **Algorithm Details(pseudocode or steps)(both searching and Sorting):**
- **For Searching/Sorting: Courses_display_pseudocode**
- **File name(for storing the details)**
- **File name to be used is:-courses_setting .txt**

Course : Sorting Algorithm used

- **Bubble sort**:
- Algorithm:
- Step 1: Start at the beginning of the list.
- Step 2:Compare the current element with the next element.
- Step 3: If the current element is greater than the next, swap them.
- Step 4: Move to the next element and repeat step 2.
- Step 5:Continue through the list until you reach the end. One pass will "bubble" the largest element to the end.
- Step 6:Repeat the above steps, ignoring the last sorted elements, until no more swaps are needed.

SRM UNIVERSITY AP
Amaravati

Course : Comparison of Sorting Algorithm

- **Selection sort:**
- Algorithm:
- Step 1: Divide the array into two parts: a sorted portion and an unsorted portion.
- Step 2: Start with the entire array as unsorted.
- Step 3: For each position in the array:
- -Find the minimum element from the unsorted portion.
- -Swap this minimum element with the element at the current position (adding it to the sorted portion).
- Step 4: Move the boundary between the sorted and unsorted portions one element to the right.
- Step 5:Repeat until the unsorted portion is empty.

Course : Time Complexity of Sorting Algorithm

SI.NO	Algorithm name	Compared Algorithm
1	Bubble sort	O(n)
2	Selection sort	O(n^2)

Course : Searching Algorithm used

- **Algorithm Name: linear search**
- **Algorithm:**
  - 1. Start at the first element of the array or list.
  - 2. Compare the target element (the value being searched for) with the current element.
  - 3. If a match is found, return the index of the current element and terminate the search.
  - 4. If no match is found, move to the next element in the array or list.
  - 5. Repeat steps 2-4 until the end of the array or list is reached.
  - 6. If the target element is not found, return a result indicating "element not found" (often -1).

Course : Comparison of Searching Algorithm

- **Algorithm used for comparison: Binary search**
- **Algorithm**
  - Here are the steps for the Binary Search algorithm:
  - 1. Sort the array if it is not already sorted.
  - 2. Initialize two pointers: left (start of the array) and right (end of the array).
  - 3. While left is less than or equal to right:
  - Calculate the midpoint as mid = (left + right) / 2.
  - Compare the target element with the element at mid.
  - 4. If the target element matches the element at mid, return the index mid.
  - 5. If the target element is less than the element at mid, move right to mid - 1.
  - 6. If the target element is greater than the element at mid, move left to mid + 1.
  - 7. Repeat steps 3–6 until the target is found or left exceeds right.
  - 8. If the target element is not found, return "element not found" (often -1).

Course : Time Complexity of Searching Algorithm

Search Type	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(logn)	O(long)

Sample Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort used in binary search example

// Bubble Sort
void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}
void selectionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
```

```
minIndex = j;
            }
        }
        std::swap(arr[i], arr[minIndex]);
    }
}

// Linear Search
int linearSearch(const std::vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target) {
            return i; // return the index of the target element
        }
    }
    return -1; // element not found
}

// Binary Search (assumes array is sorted)
int binarySearch(const std::vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid; // target found
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
```

```
}
    }
    return -1; // element not found
}

// Helper function to print the array
void printArray(const std::vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> arr = {64, 25, 12, 22, 11};
    int target = 22;

    // Bubble Sort
    std::vector<int> bubbleArr = arr;
    std::cout << "Original array for Bubble Sort: ";
    printArray(bubbleArr);
    bubbleSort(bubbleArr);
    std::cout << "Sorted array using Bubble Sort: ";
    printArray(bubbleArr);
```

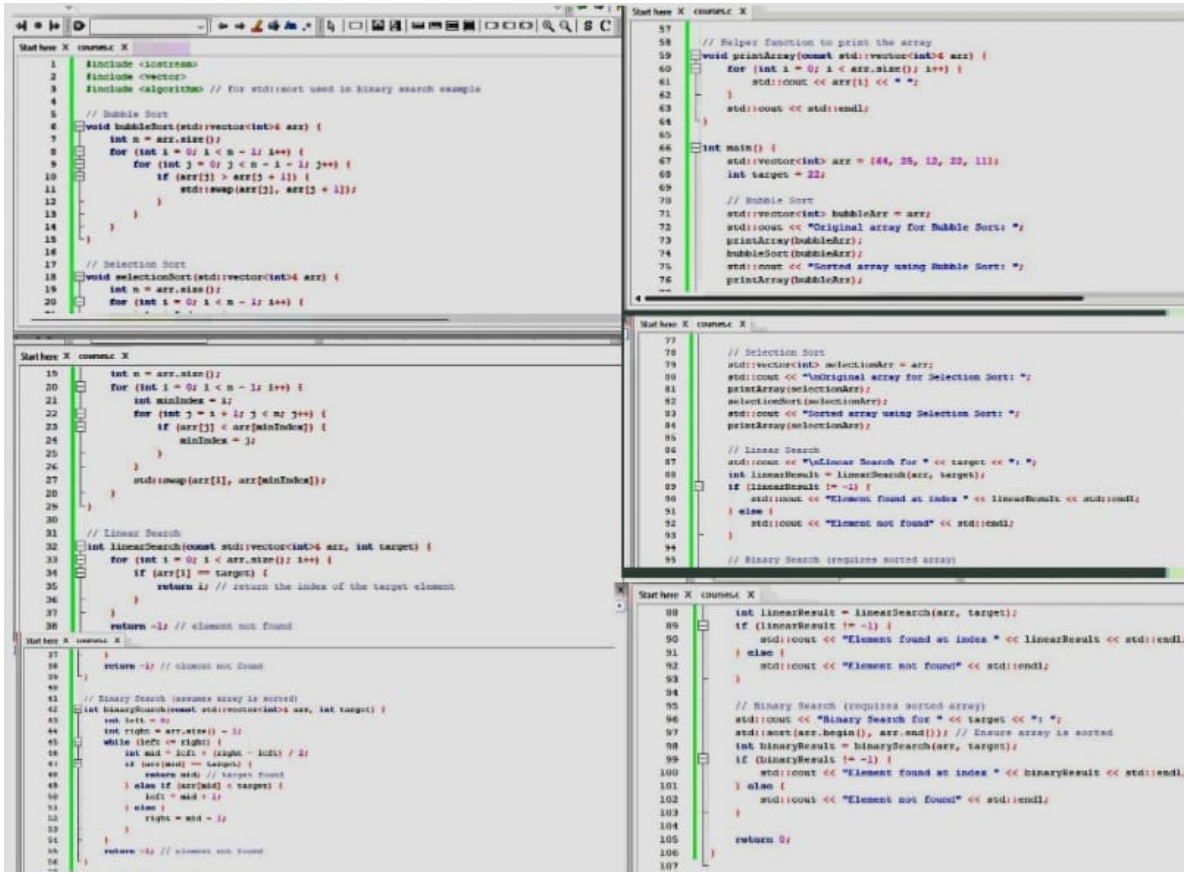
```
// Selection Sort
    std::vector<int> selectionArr = arr;
    std::cout << "\nOriginal array for Selection Sort: ";
    printArray(selectionArr);
    selectionSort(selectionArr);
    std::cout << "Sorted array using Selection Sort: ";
    printArray(selectionArr);

    // Linear Search
    std::cout << "\nLinear Search for " << target << ": ";
    int linearResult = linearSearch(arr, target);
    if (linearResult != -1) {
        std::cout << "Element found at index " << linearResult << std::endl;
    } else {
        std::cout << "Element not found" << std::endl;
    }

    // Binary Search (requires sorted array)
    std::cout << "Binary Search for " << target << ": ";
    std::sort(arr.begin(), arr.end()); // Ensure array is sorted
    int binaryResult = binarySearch(arr, target);
    if (binaryResult != -1) {
        std::cout << "Element found at index " << binaryResult << std::endl;
    } else {
        std::cout << "Element not found" << std::endl;
    }

    return 0;
```

Sample Screen Shots:



SRM UNIVERSITY AP
Amaravati

Conclusion:

The Program Management module provides an effective solution for managing program records, with functionalities for CRUD operations and efficient data retrieval via sorting and searching. By comparing sorting and searching algorithms, this project demonstrates the importance of choosing the right algorithm for performance optimization. The module's modular structure, alongside file-based data persistence, ensures maintainable system for program management.

SRM UNIVERSITY AP
Amaravati

Thank You