OBE IMPLEMENTATION: COURSE OUTCOME SETTING

BY

**TEAM NAME:** KANYA RAASI

**M.NOWNEETH REDDY (AP23110011082)**

**P. VENKAT (AP23110011119)**

**S.SHANMUK (AP23110011102)**

**S.SANTHOSH (AP23110011098)**

**A.RAKESH (AP23110011078)**

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



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**SRM UNIVERSITY AP::AMARAVATI**

**INDEX**

**Introduction**

Project Modules:

**Architecture Diagram**

**Module Description**

Programming Details naming conventions to be used:

Field/table details

Algorithm Details:

(i)Sorting

(ii)Searching

(ii) Storing the details in a text file

**Source Code**

**Comparison of Sorting Algorithms**

**Comparison of Searching Algorithms**

**Screen Shots**

**Conclusion**

**INTRODUCTION**

As part of the implementation of Outcome-Based Education (OBE) at SRM-AP, you are tasked with developing an application in a programming language of your choice that incorporates searching and sorting algorithms. The project will focus on **Insertion Sort** and **Merge Sort.**

**Insertion Sort** is a straightforward algorithm that sorts an array by building a sorted section one element at a time, with a worst-case time complexity of O(n2 ) and a best-case complexity of O(n) when the data is nearly sorted. Its advantages include simplicity and efficiency for small datasets, while its disadvantages are inefficiency on larger lists.

**Merge Sort**, in contrast, is a more advanced divide-and-conquer algorithm that consistently achieves a time complexity of O(nlog n). It is efficient for larger datasets and maintains stability, but it requires additional space for merging, resulting in a space complexity of O(n).

In comparing these algorithms, Insertion Sort is optimal for small datasets due to its low overhead, while Merge Sort excels in handling larger datasets efficiently. This project will provide students with practical insights into sorting algorithms, enhancing their understanding of algorithm efficiency within the OBE framework at SRM-AP.

**Project Modules:**

Various Modules available in the project are

1.Blooms Level setting

2.Program Level Objective Setting

3.University

4.Schools

5.Department

6.Programs

7.Courses

8.Course objective setting

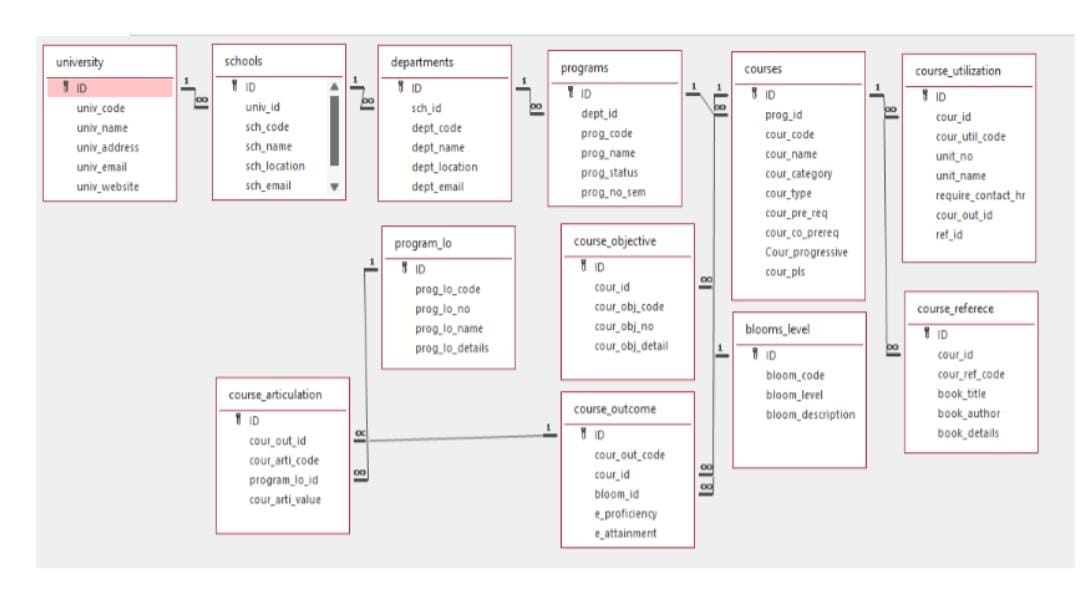
**9.Course Outcome Setting**

10.Course Articulation matrix Setting

11.Course Utilization Setting

12.Course Reference Setting.

**ARCHITECTURE DIAGRAM:**



**Module Description**

**Module Name**: COURSE OUTCOME SETTING

**Module Description:**

This module is designed to implement sorting algorithms, notably Insertion Sort and Merge Sort, to manage course reference data effectively. Along with features for searching and sorting according to predetermined algorithms, the program will let users create, edit, retrieve, and remove course details. This module will handle the following fields:

|  |  |
| --- | --- |
| **FIELD NAME** | **DATA TYPE** |
| ID | integer |
| Cour\_out\_code | string |
| Cour\_id | string |
| Bloom\_id | string |
| e\_proficiency | string |
| e\_attainment | string |

**Programming Details Naming Conventions to be Used:**

**File Name:** course\_outcome\_setting.txt

**Function/Method Names:**

1.Create: course\_outcome\_create

2.Update: course\_outcome\_update

3.Retrieve: course\_outcome\_retrieve

4.Delete: course\_outcome\_delete

5.Sorting: course\_outcome\_sorting

6. course\_outcome\_search

7.Storing: course\_outcome\_storing

8.Comparison Functions (Both Searching and Sorting):

* **For Searching:**

Linear Search: course\_outcome\_compare\_search\_linear

Binary Search: course\_outcome\_compare\_search\_binary

* **For Sorting:**

Insertion Sort: course\_outcome\_compare\_sorting\_insertion

Merge Sort: course\_outcome\_compare\_sorting\_merge

Time Complexity Functions (Both Searching and Sorting):

* **For Searching:**

Time Complexity: course\_outcome\_complexity\_search

* **For Sorting:**

Time Complexity: course\_outcome\_complexity\_sorting

Algorithm Details (Both Searching and Sorting):

* **For Searching:**

Linear Search Algorithm: course\_outcome\_algo\_linear\_search

Binary Search Algorithm: course\_outcome\_algo\_binary\_search

* **For Sorting:**

Insertion Sort Algorithm: course\_outcome\_algo\_insertion\_sort

Merge Sort Algorithm: course\_outcome\_algo\_merge\_sort

* **File name** - File name to be used is course outcome\_setting.txt

Field/table details:

Field Name Data

type

Id

Course outcome \_cour\_out\_code

Course outcome \_cour\_id

Course outcome \_bloom\_id

Course outcome \_e\_proficiency

Course outcome\_e\_attainment

**Algorithm Details:**

(i) Sorting

Insertion Sort and Merge Sort

**Algorithm Overview**

**Insertion Sort:** This algorithm builds a sorted array one element at a time by repeatedly taking the next element from the unsorted portion and inserting it into the correct position in the sorted portion.

**Merge Sort:** This algorithm follows a divide-and-conquer approach. It divides the array into halves, recursively sorts each half, and then merges the sorted halves back together.

**Time Complexity**

**Insertion Sort:**

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

Average Case: O(n2)

Worst Case: O(n2) (when the array is sorted in reverse order)

**Merge Sort:**

Best Case: O(nlog n)

Average Case: O(n logn)

Worst Case: O(nlog n)

**Space Complexity**

**Insertion Sort:** O(1) since it is an in-place sorting algorithm.

**Merge Sort:** O(n) due to the need for temporary arrays to hold the merged results.

**Stability**

**Insertion Sort:** It is a stable sort, maintaining the relative order of equal elements.

**Merge Sort:** It is also stable, preserving the order of equal elements during merging.

**Implementation Complexity**

**Insertion Sort**: Generally easier to implement due to its straightforward logic.

**Merge Sort:** More complex to implement because of its recursive nature and merging process.

**Performance on Small Data Sets**

Both algorithms can perform well on small datasets, but Insertion Sort is often faster for small arrays due to lower overhead.

**Number of Comparisons and Swaps**

**Insertion Sort**: May require many comparisons and shifts, especially in worst-case scenarios.

**Merge Sort:** Performs a consistent number of comparisons across all cases due to its divide-and-conquer approach.

**Use Cases**

**Insertion Sort:** Suitable for small datasets or when data is mostly sorted; often used for online sorting (where data arrives in a stream).

**Merge Sort:** Preferred for larger datasets and linked lists, especially when stability is required.

**Summary**

Insertion Sort is simple and efficient for small or nearly sorted datasets due to its linear best-case performance and stability. Merge Sort, while more complex, offers consistent performance on larger datasets with its O(nlogn) time complexity and stability, making it suitable for a broader range of applications.

**(ii) Searching**

**Linear Search and Binary Search**

**Algorithm Overview**

**Linear Search:** This algorithm checks each element in the list sequentially until the desired element is found or the end of the list is reached. It can be applied to both sorted and unsorted lists.

**Binary Search:** This algorithm operates on sorted lists by repeatedly dividing the list in half. It compares the target element to the middle element, eliminating half of the remaining elements from consideration.

**Time Complexity**

**Linear Search:**

Best Case: O(1) (when the target is the first element)

Average Case: O(n)

Worst Case: O(n) (when the target is not present)

**Binary Search:**

Best Case: O(1) (when the target is at the middle)

Average Case: O(logn)

Worst Case: O(logn) (when the target is not present)

**Space Complexity**

**Linear Search:** O(1) as it requires a constant amount of space.

**Binary Search**: O(1) for iterative implementation or O(logn) for recursive implementation due to call stack usage.

**Stability**

Stability is not applicable for both algorithms since they do not rearrange elements.

**Implementation Complexity**

**Linear Search**: Simple to implement with straightforward logic.

**Binary Search:** More complex due to the need for sorting (if not already sorted) and implementing recursive or iterative logic.

**Use Cases**

**Linear Search:** Suitable for small or unsorted datasets, particularly when performance is not a primary concern.

**Binary Search**: Efficient for large, sorted datasets where quick search times are required.

**Performance**

**Linear Search:** Slower for large datasets due to its linear traversal nature.

**Binary Search:** Much faster for large datasets due to its logarithmic reduction in search space.

**Best Practices**

Use Linear Search when simplicity is more important than performance, or when dealing with small or unsorted lists.

Use Binary Search when dealing with large, sorted datasets for optimal performance.

**Summary**

In summary, Linear Search is a straightforward algorithm suitable for small or unsorted datasets, while Binary Search is significantly more efficient for searching in large, sorted datasets. The choice between the two depends on specific application requirements, including dataset size and whether it is sorted.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

typedef struct {

int id;

char cour\_obj\_code[15];

char cour\_id[15];

char bloom\_id[10];

float e\_proficiency;

float e\_attainment;

} CourseOutcome;

CourseOutcome course\_outcomes[MAX];

int course\_outcome\_count = 0;

const char\* FILE\_NAME = "course\_outcomes.txt";

void load\_from\_file();

void save\_to\_file();

void create\_course\_outcome();

void update\_course\_outcome();

void retrieve\_course\_outcomes();

void delete\_course\_outcome();

void search\_course\_outcome\_by\_code();

void insertion\_sort(CourseOutcome arr[], int n);

void merge(CourseOutcome arr[], int left, int mid, int right);

void merge\_sort(CourseOutcome arr[], int left, int right);

void sort\_course\_outcomes();

void load\_from\_file() {

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

if (file == NULL) {

return; // No file exists yet

}

course\_outcome\_count = 0;

while (fscanf(file, "%d %s %s %s %f %f\n", &course\_outcomes[course\_outcome\_count].id,

course\_outcomes[course\_outcome\_count].cour\_obj\_code,

course\_outcomes[course\_outcome\_count].cour\_id,

course\_outcomes[course\_outcome\_count].bloom\_id,

&course\_outcomes[course\_outcome\_count].e\_proficiency,

&course\_outcomes[course\_outcome\_count].e\_attainment) == 6) {

course\_outcome\_count++;

}

fclose(file);

}

void save\_to\_file() {

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

if (file == NULL) {

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

return;

}

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

fprintf(file, "%d %s %s %s %.2f %.2f\n", course\_outcomes[i].id,

course\_outcomes[i].cour\_obj\_code,

course\_outcomes[i].cour\_id,

course\_outcomes[i].bloom\_id,

course\_outcomes[i].e\_proficiency,

course\_outcomes[i].e\_attainment);

}

fclose(file);

}

void create\_course\_outcome() {

if (course\_outcome\_count >= MAX) {

printf("Course outcome list is full!\n");

return;

}

CourseOutcome co;

printf("Enter Course Outcome ID: ");

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

printf("Enter Course Objective Code: ");

scanf("%s", co.cour\_obj\_code);

printf("Enter Course ID: ");

scanf("%s", co.cour\_id);

printf("Enter Bloom ID: ");

scanf("%s", co.bloom\_id);

printf("Enter Expected Proficiency: ");

scanf("%f", &co.e\_proficiency);

printf("Enter Expected Attainment: ");

scanf("%f", &co.e\_attainment);

course\_outcomes[course\_outcome\_count++] = co;

save\_to\_file();

printf("Course outcome created successfully!\n");

}

void update\_course\_outcome() {

int id;

printf("Enter Course Outcome ID to update: ");

scanf("%d", &id);

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

if (course\_outcomes[i].id == id) {

printf("Enter new Course Objective Code: ");

scanf("%s", course\_outcomes[i].cour\_obj\_code);

printf("Enter new Course ID: ");

scanf("%s", course\_outcomes[i].cour\_id);

printf("Enter new Bloom ID: ");

scanf("%s", course\_outcomes[i].bloom\_id);

printf("Enter new Expected Proficiency: ");

scanf("%f", &course\_outcomes[i].e\_proficiency);

printf("Enter new Expected Attainment: ");

scanf("%f", &course\_outcomes[i].e\_attainment);

save\_to\_file();

printf("Course outcome updated successfully!\n");

return;

}

}

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

}

void retrieve\_course\_outcomes() {

printf("\nList of Course Outcomes:\n");

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

printf("ID: %d\nCourse Objective Code: %s\nCourse ID: %s\nBloom ID: %s\nExpected Proficiency: %.2f\nExpected Attainment: %.2f\n\n",

course\_outcomes[i].id,

course\_outcomes[i].cour\_obj\_code,

course\_outcomes[i].cour\_id,

course\_outcomes[i].bloom\_id,

course\_outcomes[i].e\_proficiency,

course\_outcomes[i].e\_attainment);

}

}

void delete\_course\_outcome() {

int id;

printf("Enter Course Outcome ID to delete: ");

scanf("%d", &id);

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

if (course\_outcomes[i].id == id) {

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

course\_outcomes[j] = course\_outcomes[j + 1];

}

course\_outcome\_count--;

save\_to\_file();

printf("Course outcome deleted successfully!\n");

return;

}

}

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

}

void search\_course\_outcome\_by\_code() {

char code[15];

printf("Enter Course Objective Code to search: ");

scanf("%s", code);

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

if (strcmp(course\_outcomes[i].cour\_obj\_code, code) == 0) {

printf("ID: %d\nCourse Objective Code: %s\nCourse ID: %s\nBloom ID: %s\nExpected Proficiency: %.2f\nExpected Attainment: %.2f\n\n",

course\_outcomes[i].id,

course\_outcomes[i].cour\_obj\_code,

course\_outcomes[i].cour\_id,

course\_outcomes[i].bloom\_id,

course\_outcomes[i].e\_proficiency,

course\_outcomes[i].e\_attainment);

return;

}

}

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

}

void insertion\_sort(CourseOutcome arr[], int n) {

for (int i = 1; i < n; i++) {

CourseOutcome key = arr[i];

int j = i - 1;

while (j >= 0 && strcmp(arr[j].cour\_obj\_code, key.cour\_obj\_code) > 0) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

printf("Course outcomes sorted by Course Objective Code using Insertion Sort!\n");

}

void merge(CourseOutcome arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

CourseOutcome \*L = malloc(n1 \* sizeof(CourseOutcome));

CourseOutcome \*R = malloc(n2 \* sizeof(CourseOutcome));

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

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

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

R[j] = arr[mid + 1 + j];

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

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

if (strcmp(L[i].cour\_obj\_code, R[j].cour\_obj\_code) <= 0) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

free(L);

free(R);

}

void merge\_sort(CourseOutcome arr[], int left, int right) {

if (left < right) {

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

merge\_sort(arr, left, mid); // Sort first half

merge\_sort(arr, mid + 1, right); // Sort second half

merge(arr, left, mid, right); // Merge sorted halves

}

}

void sort\_course\_outcomes() {

int choice;

printf("\nChoose sorting algorithm:\n");

printf("1. Insertion Sort\n");

printf("2. Merge Sort\n");

scanf("%d", &choice);

switch(choice) {

case 1:

insertion\_sort(course\_outcomes, course\_outcome\_count);

retrieve\_course\_outcomes(); // Display sorted outcomes

break;

case 2:

merge\_sort(course\_outcomes, 0, course\_outcome\_count - 1);

printf("Course outcomes sorted by Course Objective Code using Merge Sort!\n");

retrieve\_course\_outcomes(); // Display sorted outcomes

break;

default:

printf("Invalid choice!\n");

break;

}

}

int main() {

load\_from\_file();

int choice;

while (1) {

printf("\n1. Create Course Outcome\n2. Update Course Outcome\n3. Retrieve Course Outcomes\n4. Delete Course Outcome\n5. Search by Code\n6. Sort by Code\n7. Exit\n");

printf("Enter your choice: ");

if(scanf("%d", &choice) != 1){

fprintf(stderr,"Invalid input! Exiting...\n");

exit(EXIT\_FAILURE);

}

switch (choice) {

case 1:

create\_course\_outcome();

break;

case 2:

update\_course\_outcome();

break;

case 3:

retrieve\_course\_outcomes();

break;

case 4:

delete\_course\_outcome();

break;

case 5:

search\_course\_outcome\_by\_code();

break;

case 6:

sort\_course\_outcomes();

break;

case 7:

exit(0);

default:

printf("Invalid choice!\n");

}

}

return 0;

}

**Comparison of Sorting Algorithms:**

**1. Insertion Sort** (Primary Algorithm):

**Advantage:** Simple to understand and implement; efficient for small datasets and adaptive for partially sorted data.

**Disadvantage:** Poor performance for larger datasets due to O(n²) time complexity.

**Algorithm for Insertion Sort():**

function insertionSort(arr)

for i from 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

return arr

**Time complexity:**

**Insertion sort:** o(n^2)

**2. Merge sort**(Comparison Algorithm):

**Advantage:** Merge Sort is efficient and stable for larger datasets.

**Disadvantage:** It is slower for smaller datasets, and it uses more

space.

**Algorithm for Merge Sort():**

function mergeSort(arr)

if length(arr) <= 1

return arr

mid = length(arr) / 2

left = mergeSort(arr[0 to mid-1])

right = mergeSort(arr[mid to end])

return merge(left, right)

function merge(left, right)

result = empty array

while left is not empty and right is not empty

if left[0] <= right[0]

append left[0] to result

remove left[0] from left

else

append right[0] to result

remove right[0] from right

while left is not empty

append left[0] to result

remove left[0] from left

while right is not empty

append right[0] to result

remove right[0] from right

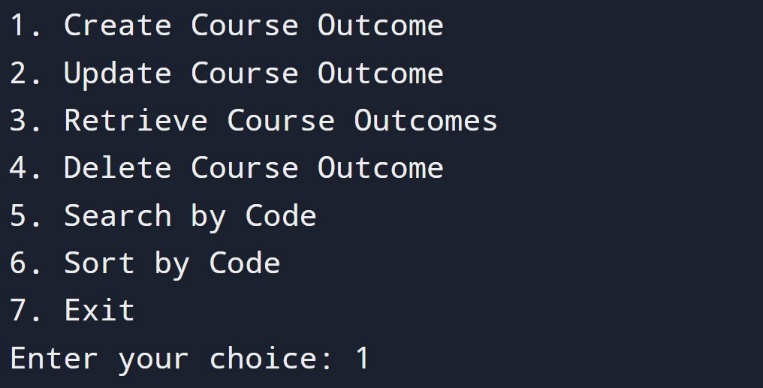
return result

**Time Complexity**:

**Merge Sort:** O(nlogn)

**SCREEN SHORTS**

**Output:**



**Create:**

A blue screen with white text

Description automatically generated

**Update:**

A blue screen with white text

Description automatically generated

**Retrieve:**

A blue screen with white text

Description automatically generated

**Delete:**

A blue screen with white text

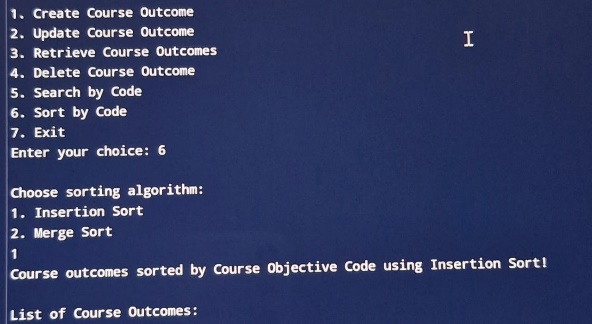
Description automatically generated

**Search by code:**

A blue screen with white text

Description automatically generated

**Sort by code:**



**Conclusion:**

Upon performing CRUD (Create, Retrieve, Update, Delete) operations, the "Source Outcome Setting" module demonstrates its effectiveness as a reliable system for managing outcome data within the project. The implementation of these operations ensures that users can effortlessly add new outcomes, access and review existing data, modify records as needed, and remove outdated or incorrect entries. To enhance this functionality, improvements in the insertion process and the implementation of a merge sort algorithm are proposed. The current insertion method allows users to add new outcomes easily, but integrating validation checks will ensure that outcome names are unique and formatted correctly. Additionally, introducing a batch insertion feature for multiple outcomes via CSV upload will save time and improve efficiency. Enhancing the user interface to provide guidance during the insertion process will further elevate user experience.

Sorting is essential for maintaining organized data, especially with larger datasets. Implementing a merge sort algorithm will provide significant benefits: it has a time complexity of O(n log n), making it well-suited for large datasets, and it preserves the relative order of records with equal keys, which is beneficial for sorting by multiple criteria. The divide and conquer approach of merge sort breaks down the dataset into smaller sublists, sorts them, and merges them back together, ensuring an organized structure. The successful execution of these CRUD operations highlights the module’s capacity to streamline data management while improving data integrity. By enhancing the insertion process and implementing a merge sort algorithm, the "Source Outcome Setting" module stands to further support efficient data handling practices within an academic or organizational framework. This practical application of essential data handling principles showcases proficiency in managing structured data effectively.