**OBE IMPLEMENTATION: UNIVERSITY SETTING**

*by*

**Lokeshwar Reddy Otikunda [AP23110011150]**

**Sri Akshaya Kodali [AP23110011163 ]**

**Sumana Sree Karani [AP23110011203]**

**Mehrosh Sultana [AP23110011212]**

**Phani Sree Ramidi [AP23110011213]**

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



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**SRM UNIVERSITY AP::AMARAVATI**

**INDEX**

[**Introduction**](#_gjdgxs) **3**

[Project Module](#_30j0zll) 3

[**Architecture Diagram**](#_1fob9te) **4**

[**Module Description**](#_3znysh7) **5**

[Programming Details naming conventions to be used:](#_2et92p0) 5

[Field/table details:](#_tyjcwt) 6

[Algorithm Details:](#_3dy6vkm) 6

[(i)Sorting](#_1t3h5sf) 6

[(ii)Searching](#_4d34og8) 7

[(ii) Storing the details in a text file](#_2s8eyo1) 8

[**Source Code**](#_17dp8vu) **9**

[**Comparison of Sorting Algorithms**](#_1ci93xb) **24**

[**Comparison of Searching Algorithms**](#_3whwml4) **25**

[**Screen Shots**](#_2bn6wsx) **27**

[**Conclusion**](#_qsh70q) **31**

# 

# Introduction

For SRM-AP's OBE implementation, We are assigned a task “University Setting” that uses efficient sorting and searching algorithms. In this project we will implement a fast sorting algorithm like Merge Sort and compare it with Quick Sort. Additionally, we will use Linear Search for data retrieval. By analyzing and displaying the time complexity of both sets of algorithms, we will demonstrate how the custom algorithms offer better performance

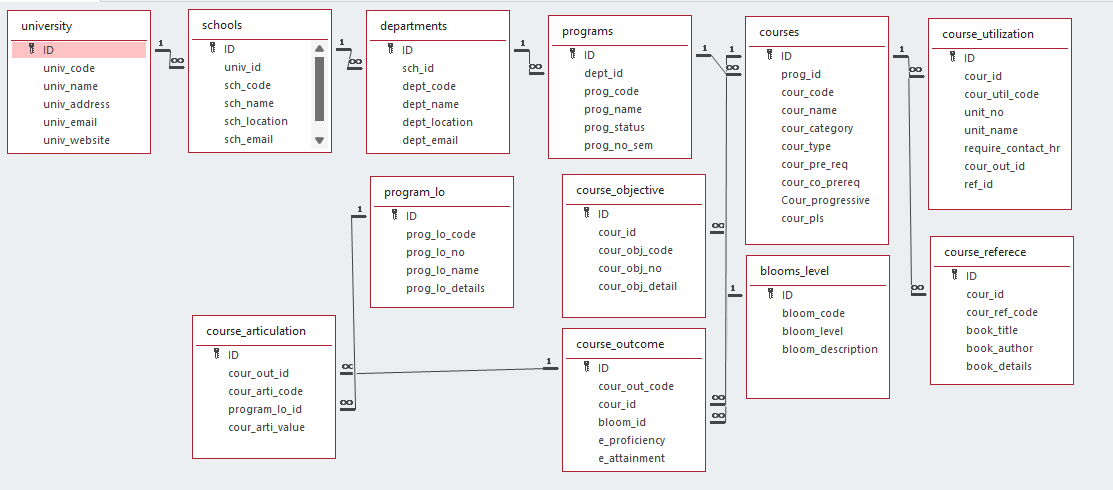
## 

## Project Module: University

# 

# 

# Architecture Diagram

****

# Module Description

**Module Name:** University

**Module Description:**

In this Module,users are allowed to **create, update, retrieve, delete, sort**, and **search** university records.

* It stores university details (code, name, address, email, website) in a text file for persistence.
* This module supports sorting and searching based on university code
* Simple file-based operations ensure easy management of university information.
* Additionally we provide the pseudo codes and time complexity analysis of sorting and searching algorithms, comparison of those algorithms with existing algorithms.

Programming Details naming conventions to be used:

* **File name:**VirtualThinkers\_university\_Sumana
* **Function/method name**
  + **Create:**VirtualThinkers\_university\_create
  + **Update:**VirtualThinkers\_university\_update
  + **Retrieve:**VirtualThinkers\_university\_retrive
  + **Delete:**VirtualThinkers\_university\_delete
  + **Sorting:**VirtualThinkers\_university\_MergeSort
  + **Searching:**VirtualThinkers\_university\_LinearSearch
  + **Storing:**VirtualThinkers\_university\_storing
  + **Comparison(both searching and Sorting)**:
    - For Searching-VirtualThinkers\_university\_Compare\_Search\_LinearSearch
    - For Sorting-VirtualThinkers\_university\_Compare\_Sorting\_MergeSort
  + **Time Complexity(both searching and Sorting):**
    - For Searching-VirtualThinkers\_university\_complexity\_Search
    - For Sorting-VirtualThinkers\_university\_compexity\_Sorting
  + **Algorithm Details(pseudocode or steps)(both searching and Sorting):**
    - For Searching-VirtualThinkers\_university\_LinearSearch\_details
    - For Sorting-VirtualThinkers\_university\_MergeSort\_details
* **File name(for storing the details)**
  + File name to be used is:-university\_setting.txt

## 

## 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:

### (i)Sorting

* sorting based on **university code ,university\_name , university\_email.**
* Compare the algorithm you have used with any of the other sorting algorithm

**Approach:**

**Merge Sort:** Divide and Conquer (Divides the array into two halves and merges them).

**Quick Sort:** Divide and Conquer (Partitions the array around a pivot element).

**Stability:**

**Merge Sort:** Stable (Preserves the order of equal elements).

**Quick Sort:** Unstable (May not preserve the order of equal elements).

* Display the time complexity of both algorithms.

**Best Case Time Complexity: Merge Sort:** O(n log n).**Quick Sort:** O(n log n).

**Average Case Time Complexity: Merge Sort:** O(n log n) **Quick Sort:** O(n log n).

**Worst Case Time Complexity: Merge Sort:** O(n log n) **Quick Sort:** O(n²) (Occurs when the pivot is poorly chosen, e.g., if the array is already sorted).

* Displaying the pseudocode/algorithm of the sorting algorithm

function mergeSort(arr):");

if the length of arr is 1 or less

return arr

mid = the middle index of arr

left = mergeSort(arr[:mid])

right = mergeSort(arr[mid:])

return merge(left, right)

function merge(left, right)

result = []

while both left and right are 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

append any remaining elements from left and right to result

return result

### (ii)Searching

* searching based on **university code,university\_name,university\_email**
* Comparing the algorithm used with any of the other algorithm

**Approach:**

**Linear Search**: It is an iterative search method that checks each element one by one, starting from the beginning of the list, until it finds the target or reaches the end.

**Binary Search**: It is a divide-and-conquer algorithm that works by repeatedly dividing a sorted list into two halves. It compares the target value with the middle element and eliminates half of the remaining elements in each step.

**Data Requirement:**

**Linear Search**: Can be used on **both sorted and unsorted data**. It does not require any specific ordering of the elements.

**Binary Search**: **Requires sorted data**. The list must be ordered (either ascending or descending) for binary search to function correctly.

* Display the time complexity of both algorithms.

**Best Case Time Complexity:Linear Search**: **O(1) Binary Search**: **O(1)**.

**Average Case Time Complexity: Linear Search**:**O(n)**, **Binary Search**: **O(log n)**

**Worst Case Time Complexity: Linear Search**: **O(n)** **Binary Search**: **O(log n)**.

* Displaying the pseudocode/algorithm of the searching algorithm

function linearSearch(arr, target)

for each element in arr

if element == target

return the index of element

return -1 (if target is not found)

### 

### 

### (ii) Storing the details in a text file

* Storing the details in the text file once details are entered.

**1. Open file** in append mode (fopen("university\_setting.txt", "a")).

If the file can't be opened, display an error and exit.

**2. Write university data** (id,code, name, address, email, website) to the file using fprintf().

**3. Close the file** using fclose()

**Display success message**: "University is added and saved successfully!"

* Delete the detail from the text file once details are deleted.

1. Open university\_setting.txt in read mode.

2. If the file can't be opened, print an error and exit.

3. Initialize an array `temp\_u` for university data and variables `temp\_count` (for counting) and `found` (as a flag).

4. Read university data (id,code, name, address, email, website) into `temp\_u` until EOF.

5. If `univ\_code` matches the target, set `found` to 1 and skip adding it to `temp\_u`.

6. Close the file.

7. If `found` is 0, print "University not found" and exit.

* Update the text file once details are updated.

**1. Open file** in write mode (fopen("university\_setting.txt", "w")):

If the file can't be opened, display an error and exit.

**2. Loop through all university records** (for (int j = 0; j < count; j++)):

For each university, write its details (id,code, name, address, email, website) to the file using fprintf().

**3. Close the file** using fclose().

**Display success message**: "University details are updated and saved successfully!"

# Source Code

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_UNIVERSITIES 100

// Structure to handle University details

struct University {

int ID;

char univ\_code[10];

char univ\_name[10];

char univ\_address[10];

char univ\_email[15];

char univ\_website[20];

};

//Function Declarations

void VirtualThinkers\_university\_create(struct University u[], int \*count);

void VirtualThinkers\_university\_update(struct University u[], int count);

void VirtualThinkers\_university\_retrieve(struct University u[], int \*count);

void VirtualThinkers\_university\_delete(struct University u[], int \*count);

void VirtualThinkers\_university\_MergeSort(struct University u[], int left, int right, int sort\_by);

void merge(struct University arr[], int left, int right, int sort\_by);

void VirtualThinkers\_university\_storing(struct University u[], int \*count, int mode, int univ\_code);

void VirtualThinkers\_university\_displayMenu();

void VirtualThinkers\_university\_LinearSearch(struct University u[], int count);

// Function to handle all file operations

void VirtualThinkers\_university\_storing(struct University u[], int \*count, int mode, int ID) {

FILE \*file = NULL;

// Writing new university details into the file (create or update)

if (mode == 1) {

file = fopen("university\_setting.txt", "w");

if (file == NULL) {

printf("File cannot be opened for writing.\n");

return;

}

for (int i = 0; i < \*count; i++) {

fprintf(file, "%d\n%s\n%s\n%s\n%s\n%s\n", u[i].ID,u[i].univ\_code, u[i].univ\_name,

u[i].univ\_address, u[i].univ\_email, u[i].univ\_website);

}

fclose(file);

printf("University data is saved successfully into the file!\n");

}

// Deleting selected university from the file

else if (mode == 3) {

file = fopen("university\_setting.txt", "r");

if (file == NULL) {

printf("The file cannot be opened for reading.\n");

return;

}

struct University temp\_u[MAX\_UNIVERSITIES];

int temp\_count = 0;

int found = 0;

// Reading all universities into a temporary array

while (fscanf(file, "%d\n", &temp\_u[temp\_count].ID) == 1) {

fscanf(file, "%[^\n]\n", temp\_u[temp\_count].univ\_code);

fscanf(file, "%[^\n]\n", temp\_u[temp\_count].univ\_name);

fscanf(file, "%[^\n]\n", temp\_u[temp\_count].univ\_address);

fscanf(file, "%[^\n]\n", temp\_u[temp\_count].univ\_email);

fscanf(file, "%[^\n]\n", temp\_u[temp\_count].univ\_website);

if (temp\_u[temp\_count].ID == ID) {

found = 1; // if university found, mark it as 1

continue;

}

temp\_count++;

}

fclose(file);

if (!found) {

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

return;

}

// Updating the list of universities back into the file after deleting a university

\*count = temp\_count; // Updating count after deletion

file = fopen("university\_setting.txt", "w");

if (file == NULL) {

printf("The file cannot be opened for writing.\n");

return;

}

for (int i = 0; i < \*count; i++) {

fprintf(file, "%d\n%s\n%s\n%s\n%s\n%s\n",temp\_u[i].ID, temp\_u[i].univ\_code, temp\_u[i].univ\_name,

temp\_u[i].univ\_address, temp\_u[i].univ\_email, temp\_u[i].univ\_website);

}

fclose(file);

printf("University with ID %d has been deleted successfully.\n", ID);

}

}

// Create new university

void VirtualThinkers\_university\_create(struct University u[], int \*count) {

if (\*count >= MAX\_UNIVERSITIES) {

printf("Maximum Count Reached! It is not possible to add any more universities.\n");

return;

}

printf("Enter the University ID: ");

scanf("%d", &u[\*count].ID);

// Checking for same university ID

for (int i = 0; i < \*count; i++) {

if (u[i].ID == u[\*count].ID) {

printf("Error: University with ID %d already exists. Please enter a different ID.\n", u[\*count].ID);

printf("Enter a unique University ID: ");

scanf("%d", &u[\*count].ID);

i = -1;

}

}

printf("Enter the University Code : ");

scanf(" %[^\n]", u[\*count].univ\_code);

printf("Enter the University Name (3-6 characters): ");

while (1) {

scanf(" %[^\n]", u[\*count].univ\_name);

int len = strlen(u[\*count].univ\_name);

if (len >= 3 && len <= 6) break;

else printf("Error: The name of the university must consist of 3-6 characters. Please enter again: ");

}

printf("Enter the University Address (3-6 characters): ");

while (1) {

scanf(" %[^\n]", u[\*count].univ\_address);

int len = strlen(u[\*count].univ\_address);

if (len >= 3 && len <= 6) break;

else printf("Error: The address of the university must consist of 3-6 characters. Please enter again: ");

}

printf("Enter the University Email (12-13 characters): ");

while (1) {

scanf("%s", u[\*count].univ\_email);

int len = strlen(u[\*count].univ\_email);

if (len >= 12 && len <= 13) break;

else printf("Error: The email of the university must consist of 12-13 characters. Please enter again: ");

}

printf("Enter the University Website: ");

scanf("%s", u[\*count].univ\_website);

(\*count)++;

// Storing the new university details into the file (mode 1 for writing)

VirtualThinkers\_university\_storing(u, count, 1, 0);

printf("University is added and saved successfully!\n");

}

// Update existing university details

void VirtualThinkers\_university\_update(struct University u[], int count) {

int ID;

int found = 0;

if (count == 0) {

printf("There are no universities available to update.\n");

return;

}

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

scanf("%d", &ID);

for (int i = 0; i < count; i++) {

if (u[i].ID == ID) {

found = 1;

printf("Updating the University Details\n");

printf("Enter the New University Code : ");

scanf(" %[^\n]", u[i].univ\_code);

printf("Enter the New University Name (3-5 characters): ");

while (1) {

scanf(" %[^\n]", u[i].univ\_name);

int len = strlen(u[i].univ\_name);

if (len >= 3 && len <= 6) break;

else printf("Error: The name of the university must consist of 3 to 6 characters. Please enter again: ");

}

printf("Enter the New University Address (3-6 characters): ");

while (1) {

scanf(" %[^\n]", u[i].univ\_address);

int len = strlen(u[i].univ\_address);

if (len >= 3 && len <= 6) break;

else printf("Error: The address of the university must consist of 3 to 6 characters. Please enter again: ");

}

printf("Enter the New University Email (12-13 characters): ");

while (1) {

scanf(" %[^\n]", u[i].univ\_email);

int len = strlen(u[i].univ\_email);

if (len >= 12 && len <= 13) break;

else printf("Error: The email of the university must consist of 12 to 13 characters. Please enter again: ");

}

printf("Enter the New University Website: ");

scanf(" %[^\n]", u[i].univ\_website);

// Storing the updated university details into the file

VirtualThinkers\_university\_storing(u, &count, 1, 0);

printf("University details are updated and saved successfully!\n");

break;

}

}

if (!found) {

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

}

}

// Delete university

void VirtualThinkers\_university\_delete(struct University u[], int \*count) {

int ID;

if (\*count == 0) {

printf("There are no universities available to delete.\n");

return;

}

printf("Enter the ID of the university you need to delete: ");

scanf("%d", &ID);

// Loop through to find and delete university

int found = 0;

for (int i = 0; i < \*count; i++) {

if (u[i].ID == ID) {

found = 1;

for (int j = i; j < \*count - 1; j++) {

u[j] = u[j + 1];

}

(\*count)--;

break;

}

}

if (!found) {

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

return;

}

// Storing the updated list of universities after deletion

VirtualThinkers\_university\_storing(u, count, 1, 0);

printf("University with ID %d has been deleted successfully.\n", ID);

}

// Merge Sort function to sort universities based on different criteria

void merge(struct University arr[], int left, int right, int sort\_by) {

if (left >= right) return;

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

merge(arr, left, mid, sort\_by);

merge(arr, mid + 1, right, sort\_by);

int n1 = mid - left + 1;

int n2 = right - mid;

struct University L[n1], R[n2];

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) {

int compare = 0;

if (sort\_by == 1)

compare = L[i].univ\_code - R[j].univ\_code;

else if (sort\_by == 2)

compare = strcmp(L[i].univ\_name, R[j].univ\_name);

else if (sort\_by == 3)

compare = strcmp(L[i].univ\_email, R[j].univ\_email);

if (compare <= 0) {

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

} else {

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

}

}

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

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

}

void VirtualThinkers\_university\_MergeSort(struct University u[], int left, int right, int sort\_by) {

if (left >= right) return;

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

VirtualThinkers\_university\_MergeSort(u, left, mid, sort\_by);

VirtualThinkers\_university\_MergeSort(u, mid + 1, right, sort\_by);

merge(u, left, right, sort\_by);

}

void VirtualThinkers\_university\_display(struct University u[], int count) {

if (count == 0) {

printf("No universities available to display.\n");

return;

}

printf("\n\t\t\t\tList of Universities (Sorted)\n\n");

printf("--------------------------------------------------------------------------------------------------\n");

printf("\tCode\tName\t\tAddress\t\tEmail\t\tWebsite\n");

printf("--------------------------------------------------------------------------------------------------\n");

for (int i = 0; i < count; i++) {

printf("\t%s\t\t%s\t\t%s\t\t%s\t%s\n", u[i].univ\_code, u[i].univ\_name, u[i].univ\_address, u[i].univ\_email, u[i].univ\_website);

}

printf("\n");

}

void VirtualThinkers\_university\_LinearSearch(struct University u[], int count) {

if (count == 0) {

printf("No universities available to search.\n");

return;

}

int search\_by;

printf("Select search criteria:\n");

printf("1. Search by Code\n");

printf("2. Search by Name\n");

printf("3. Search by Email\n");

printf("Enter your choice: ");

scanf("%d", &search\_by);

int found = 0;

if (search\_by == 1) {

char code\_to\_search[10];

printf("Enter the code of the university to search: ");

scanf(" %[^\n]", code\_to\_search);

for (int i = 0; i < count; i++) {

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

printf("\nUniversity found:\n");

printf("CODE: %s\n", u[i].univ\_code);

printf("NAME: %s\n", u[i].univ\_name);

printf("ADDRESS: %s\n", u[i].univ\_address);

printf("EMAIL: %s\n", u[i].univ\_email);

printf("WEBSITE: %s\n", u[i].univ\_website);

found = 1;

break;

}

}

} else if (search\_by == 2) {

char name\_to\_search[10];

printf("Enter the name of the university to search: ");

scanf(" %[^\n]", name\_to\_search);

for (int i = 0; i < count; i++) {

if (strcmp(u[i].univ\_name, name\_to\_search) == 0) {

printf("\nUniversity found:\n");

printf("CODE: %s\n", u[i].univ\_code);

printf("NAME: %s\n", u[i].univ\_name);

printf("ADDRESS: %s\n", u[i].univ\_address);

printf("EMAIL: %s\n", u[i].univ\_email);

printf("WEBSITE: %s\n", u[i].univ\_website);

found = 1;

break;

}

}

} else if (search\_by == 3) {

char email\_to\_search[15];

printf("Enter the email of the university to search: ");

scanf(" %[^\n]", email\_to\_search);

for (int i = 0; i < count; i++) {

if (strcmp(u[i].univ\_email, email\_to\_search) == 0) {

printf("\nUniversity found:\n");

printf("CODE: %s\n", u[i].univ\_code);

printf("NAME: %s\n", u[i].univ\_name);

printf("ADDRESS: %s\n", u[i].univ\_address);

printf("EMAIL: %s\n", u[i].univ\_email);

printf("WEBSITE: %s\n", u[i].univ\_website);

found = 1;

break;

}

}

}

if (!found) {

printf("\nUniversity not found.\n");

}

}

void VirtualThinkers\_university\_retrieve(struct University u[], int \*count) {

FILE \*file = fopen("university\_setting.txt", "r");

if (file == NULL) {

printf("There are no universities available.\n");

return;

}

\*count = 0; // Resetting university count

while (fscanf(file, "%d\n", &u[\*count].ID) == 1) {

fscanf(file, "%[^\n]\n", u[\*count].univ\_code);

fscanf(file, "%[^\n]\n", u[\*count].univ\_name);

fscanf(file, "%[^\n]\n", u[\*count].univ\_address);

fscanf(file, "%[^\n]\n", u[\*count].univ\_email);

fscanf(file, "%[^\n]\n", u[\*count].univ\_website);

(\*count)++;

}

fclose(file);

if (\*count == 0) {

printf("There are no universities available to display.\n");

return;

}

printf("\n\t\t\t\tList of Universities\n\n");

printf("--------------------------------------------------------------------------------------------------\n");

printf("\tID\tCode\t\tName\t\tAddress\t\tEmail\t\tWebsite\n");

printf("--------------------------------------------------------------------------------------------------\n");

for (int i = 0; i < \*count; i++) {

printf("\t%d\t%s\t\t%s\t\t%s\t%s\t\t%s\n",u[i].ID,u[i].univ\_code,u[i].univ\_name,u[i].univ\_address,u[i].univ\_email,u[i].univ\_website);

}

printf("\n");

}

void VirtualThinkers\_university\_Compare\_Sorting(){

printf("\nComparision of Merge and Quick Sort\n");

printf("\nMerge Sort\n");

printf("1. Merge sort also follows the divide & conquer approach. \nit finds a mid element and divides the data continuously into 2 parts till each array has 1 element\n Then it sorts those sub arrays and merges back into a single array \n returns the sorted array.\n");

printf("2. It is a stable sorting algorithm.\n");

printf("3. It works faster in execution due to its recursive calls, there are multiple calls and list is sorted very fast.\n");

printf("4. Space complexity of Merge sort is O(n).\n");

printf("5. Works well on any size of data.\n");

printf("6. It sorts externally.\n");

printf("\n\tQuick Sort\n");

printf("1. QuickSort follows the divide & conquer approach \nthat selects an element as pivot element and divides/ partitions the array or \nlist around the picked pivot element by placing the pivot element in its correct position\n in the sorted array.\n The partition or array does not follow any ration only follows the position of pivot element in sorted array.\n");

printf("2. It is an unstable sorting algorithm.\n");

printf("3. Slow in execution due to iterative calls.\n");

printf("4. The space complexity of quicksort is O(log(n)) in the average case, and O(n) in the worst case.\n");

printf("5. It is not efficient for larger set of data.\n");

printf("6. Sorting is done internally.\n\n");

}

void VirtualThinkers\_university\_Compare\_Search(){

printf("\nComparision of Linear and binary Search\n");

printf("\n\tBinary Search\n");

printf("1. needs sorted data as input.\n");

printf("2. Only single dimension arrays/ lists can be used due to divide and conquer approach.\n");

printf("3. Binary Search follows the divide and conquer approach.\n It divides the array / list into 2 halves from the mid element, \nthen compares if the mid element is equal to the search element, if yes then return the mid index or position value. Else not equal, it checks if search element is smaller or greater than mid element. If smaller than mid, \nthen this process is repeated on first half or array / list. \nElse the search value greater than mid element, then same process repeated on second half of list / array.\n");

printf("4. Binary search is more efficient for large, ordered lists.\n");

printf("5. Faster in execution as only half of the data is to be checked and only 1 element is checked if same or not.\n");

printf("6. Space complexity of the binary search is O(1).\n");

printf("\n\tLinear Search\n");

printf("1. works the same for sorted and unsorted data.\n(ie. order of data does not affect the working of algorithm)\n");

printf("2. Multidimensional array can be used as it uses an iterative approach.\n");

printf("3. Linear search also called as the Sequential search follows an iterative approach.\n It iterates through each element of the list and compares it with the element to be searched.\n If the search element is found, it returns the position or index of the element.\n Else not found it returns a value (-1) NULL to show the element is not found.\n");

printf("4. Linear search is better for smaller lists or unordered data.\n");

printf("5. Slower in execution as iteration is done on every element by comparing each element every iteration.\n");

printf("6. Space complexity of the linear search is O(1).\n\n");

}

void VirtualThinkers\_university\_compexity\_Sorting(){

printf("Time Complexity of Sort Algorithms:\n");

printf("\n1. Quick Sort:");

printf("\nBest Case: O(n\*log(n))");

printf("\nAverage Case: O(n\*log(n))");

printf("\nWorst Case: O(n^2)");

printf("\n2. Merge Sort:");

printf("\nBest Case: O(n log n)");

printf("\nAverage Case: O(n log n)");

printf("\nWorst Case: O(n log n)\n");

}

void VirtualThinkers\_university\_compexity\_Search(){

printf("Time Complexity of Search Algorithms:\n");

printf("\n1. Linear Search:");

printf("\nBest Case: O(1)");

printf("\nAverage Case: O(n)");

printf("\nWorst Case: O(n)");

printf("\n2. Binary Search:");

printf("\nBest Case: O(1)");

printf("\nAverage Case: O(log n)");

printf("\nWorst Case: O(log n)\n");

}

void VirtualThinkers\_university\_Sort\_details() {

printf("\nMerge Sort Algorithm \nDivide and Conquer:\n");

printf("---------------------------------\n");

printf("Merge Sort is a divide and conquer algorithm that recursively divides an array into two halves,\n sorts each half, and then merges the sorted halves into a single sorted array.");

printf("\nBase Case:\n\tThe array has only 1 or no elements, the array is already sorted.\n The algorithm stops dividing when it reaches this condition.");

printf("\nRecursive Division:\n");

printf("For an array segment arr[left...right]:\n");

printf("\nFind the middle index: mid = (left + right) / 2.");

printf("\nRecursively apply Merge Sort to the first half: MergeSort(arr, left, mid).");

printf("\nRecursively apply Merge Sort to the second half: MergeSort(arr, mid + 1, right).");

printf("\n\tMerge Step:");

printf("\nAfter both halves are sorted, merge them back together in sorted order:");

printf("\nCreate Temporary Arrays:");

printf("\nCreate two temporary arrays to hold the elements of the left and right halves.");

printf("\nL[] holds elements from arr[left] to arr[mid].");

printf("\nR[] holds elements from arr[mid + 1] to arr[right].");

printf("\nInitialize Pointers:");

printf("\ni will track the current index in L[].");

printf("\nj will track the current index in R[].");

printf("\nk will track the position in the original array where the smallest element should go.");

printf("\nMerge Elements in Sorted Order:");

printf("\nWhile there are elements in both L[] and R[], compare L[i] with R[j]:");

printf("\nIf L[i] is smaller, place L[i] in arr[k] and increment i and k.");

printf("\nIf R[j] is smaller, place R[j] in arr[k] and increment j and k.");

printf("\nCopy Remaining Elements:");

printf("\nIf there are any remaining elements in L[], copy them to arr[k].");

printf("\nIf there are any remaining elements in R[], copy them to arr[k].");

printf("\n\tEnd of Recursive Calls:");

printf("\nThe recursive calls continue until all array segments have been sorted and merged,\nresulting in a fully sorted array.");

printf("\nSummary of steps:\nDivide the array into two halves.");

printf("\nSort each half recursively.\nMerge the two sorted halves into a sorted whole.");

printf("---------------------------------\n");

printf("\nQuick Sort \nDivide and Conquer:\n");

printf("----------------------------------\n");

printf("\nQuick Sort is a divide and conquer algorithm,\nwhich recursively partitions an array around a pivot");

printf("\nPlacing elements smaller than the pivot on its left and elements greater than the pivot on its right");

printf("\nBase Case:\nIf the array segment has zero or one element, it is already sorted.");

printf("\nQuick Sort stops dividing when it reaches this base case.");

printf("\nPartition Step:");

printf("\nFor an array segment arr[left...right]:\nChoose a Pivot:");

printf("\nSelect a pivot element from the array. Common choices include:\nThe first element (arr[left]).\nThe last element (arr[right]).\nThe middle element (arr[(left + right) / 2]).");

printf("\nHere, let's assume the pivot is arr[right] (the last element in the segment).");

printf("\nReorder Elements:");

printf("\nInitialize two pointers:\ni starts at left - 1 (just before the first element).\nj traverses from left to right - 1.");

printf("\nFor each element arr[j]:");

printf("\nIf arr[j] is less than or equal to the pivot, \nincrement i and swap arr[i] with arr[j] to move smaller elements to the left of the pivot.");

printf("\n\tPlace the Pivot:");

printf("\nAfter the loop, place the pivot at its correct position by swapping arr[i + 1] with arr[right].");

printf("\nThe pivot is now at index i + 1, \nwith all smaller elements on the left and all greater elements on the right.");

printf("\n\tRecursive Sort:");

printf("\nRecursively apply Quick Sort on the left and right partitions:");

printf("\nQuickSort(arr, left, i)sorts the elements on the left of the pivot.\nQuickSort(arr, i + 2, right)\nsorts the elements on the right of the pivot.");

printf("\n\tEnd of Recursive Calls:");

printf("\nThe recursive calls continue until all segments have been sorted, resulting in a fully sorted array.");

printf("\nSummary of Steps:");

printf("\nChoose a Pivot.\nPartition the array around the pivot, \nplacing smaller elements to its left and larger elements to its right.");

printf("\nRecursively Sort the left and right partitions.");

printf("----------------------------------\n");

}

void VirtualThinkers\_university\_Search\_details(){

printf("\nLinear Search Algorithm Pseudocode:\n");

printf("-----------------------------------\n");

printf("\nfunction linearSearch(arr, target):");

printf("\n for each element in arr:");

printf("\n if element == target:");

printf("\n return the index of element");

printf("\n return -1 (if target is not found)\n");

printf("-----------------------------------\n\n");

printf("\nBinary Search Algorithm Pseudocode:\n");

printf("------------------------------------\n");

printf("\nfunction binarySearch(arr, target):");

printf("\n low = 0");

printf("\n high = length of arr - 1");

printf("\n while low <= high:");

printf("\n mid = (low + high) / 2");

printf("\n if arr[mid] == target:");

printf("\n return mid");

printf("\n else if arr[mid] < target:");

printf("\n low = mid + 1");

printf("\n else:");

printf("\n high = mid - 1\n");

printf("\n return -1 (if target is not found)\n");

printf("-----------------------------------\n\n");

}

void VirtualThinkers\_university\_displayMenu() {

printf("\*\*\*\* University Module \*\*\*\*\*\*\*\*\n");

printf("1. Create New University\n");

printf("2. Update University\n");

printf("3. Retrieve University Details\n");

printf("4. Delete University Details\n");

printf("5. Sort University Details\n");

printf("6. Search University Details\n");

printf("7. Compare Sort Algorithms\n");

printf("8. Compare Search Algorithms\n");

printf("9. Display Time Complexity of Sorting Algorithm\n");

printf("10. Display Time Complexity of Searching Algorithm\n");

printf("11. Display Sort Algorithm Details\n");

printf("12. Display Search Algorithm Details\n");

printf("13. Exit Application\n");

}

int main() {

struct University uni[MAX\_UNIVERSITIES];

int count = 0;

int option,sort\_by;

do {

VirtualThinkers\_university\_displayMenu();

printf("Please select an option: ");

scanf("%d", &option);

getchar();

switch (option) {

case 1:

VirtualThinkers\_university\_create(uni, &count);

break;

case 2:

VirtualThinkers\_university\_update(uni, count);

break;

case 3:

VirtualThinkers\_university\_retrieve(uni, &count);

break;

case 4:

VirtualThinkers\_university\_delete(uni, &count);

break;

case 5:

printf("Choose sort criteria:\n1. Sort by Code\n2. Sort by Name\n3. Sort by Email\n");

scanf("%d", &sort\_by);

VirtualThinkers\_university\_MergeSort(uni, 0, count - 1, sort\_by);

printf("Universities sorted successfully.\n");

VirtualThinkers\_university\_display(uni, count); // Display sorted list

break;

case 6:

VirtualThinkers\_university\_LinearSearch(uni, count);

break;

case 7:

VirtualThinkers\_university\_Compare\_Sorting( );

break;

case 8:

VirtualThinkers\_university\_Compare\_Search( );

break;

case 9:

VirtualThinkers\_university\_compexity\_Sorting();

break;

case 10:

VirtualThinkers\_university\_compexity\_Search();

break;

case 11:

VirtualThinkers\_university\_Sort\_details();

break;

case 12:

VirtualThinkers\_university\_Search\_details();

break;

case 13:

printf("Exiting application...\n");

break;

default:

printf("Invalid option. Please enter a valid option.\n");

break;

}

} while (option != 13);

return 0;

}

# 

# Comparison of Sorting Algorithms

* **Approach:**
  + **Merge Sort:** Divide and Conquer (Divides the array into two halves and merges them).
  + **Quick Sort:** Divide and Conquer (Partitions the array around a pivot element).
* **Stability:**
  + **Merge Sort:** Stable (Preserves the order of equal elements).
  + **Quick Sort:** Unstable (May not preserve the order of equal elements).
* **Best Case Time Complexity:**
  + **Merge Sort:** O(n log n).
  + **Quick Sort:** O(n log n).
* **Average Case Time Complexity:**
  + **Merge Sort:** O(n log n).
  + **Quick Sort:** O(n log n).
* **Worst Case Time Complexity:**
  + **Merge Sort:** O(n log n).
  + **Quick Sort:** O(n²) (Occurs when the pivot is poorly chosen, e.g., if the array is already sorted).
* **Space Complexity:**
  + **Merge Sort:** O(n) (Requires extra space for merging).
  + **Quick Sort:** O(log n) (In-place sorting with recursion; O(n) in worst case due to recursion depth).
* **Efficiency:**
  + **Merge Sort:** Slower for in-memory sorting due to space overhead.
  + **Quick Sort:** Faster for in-memory sorting (on average).
* **Sorting Type:**
  + **Merge Sort:** External Sort (Suitable for large data sets that do not fit in memory).
  + **Quick Sort:** Internal Sort (Efficient for in-memory data).

# Comparison of Searching Algorithms

1. **Approach:**
   * **Linear Search**: It is an iterative search method that checks each element one by one, starting from the beginning of the list, until it finds the target or reaches the end.
   * **Binary Search**: It is a divide-and-conquer algorithm that works by repeatedly dividing a sorted list into two halves. It compares the target value with the middle element and eliminates half of the remaining elements in each step.
2. **Data Requirement:**
   * **Linear Search**: Can be used on **both sorted and unsorted data**. It does not require any specific ordering of the elements.
   * **Binary Search**: **Requires sorted data**. The list must be ordered (either ascending or descending) for binary search to function correctly.
3. **Best Case Time Complexity:**
   * **Linear Search**: The best case occurs when the target is found at the first element of the list, so the time complexity is **O(1)**.
   * **Binary Search**: The best case occurs when the target is found at the middle element of the list, so the time complexity is **O(1)**.
4. **Average Case Time Complexity:**
   * **Linear Search**: On average, linear search will check half of the elements in the list. The time complexity is **O(n)**, where n is the number of elements in the list.
   * **Binary Search**: In binary search, the list is halved each time. The average time complexity is **O(log n)**, where n is the number of elements in the list.
5. **Worst Case Time Complexity:**
   * **Linear Search**: The worst case occurs when the target is at the last position in the list (or not present at all). The time complexity is **O(n)**.
   * **Binary Search**: In the worst case, binary search divides the list down to a single element. The time complexity is **O(log n)**.
6. **Space Complexity:**
   * **Linear Search**: It uses a constant amount of extra space, so the space complexity is **O(1)**.
   * **Binary Search**: It also uses a constant amount of extra space, so the space complexity is **O(1)**.
7. **Time Efficiency:**
   * **Linear Search**: Linear search is slower for large datasets, especially when the data is unsorted, as it has to check each element sequentially.
   * **Binary Search**: Binary search is faster for large, sorted datasets because it eliminates half of the remaining elements with each comparison, making it more efficient (logarithmic time).
8. **Suitability for Sorted Data:**
   * **Linear Search**: Works well on both sorted and unsorted data. It does not require the list to be sorted.
   * **Binary Search**: Works **only on sorted data**. If the data is unsorted, binary search cannot be used.
9. **Implementation Simplicity:**
   * **Linear Search**: Simple to implement and understand, as it just iterates through the list.
   * **Binary Search**: More complex to implement, especially if the data is not sorted initially. The algorithm requires the list to be ordered and involves recursive or iterative halving of the list.
10. **Use Case:**
    * **Linear Search**: Best for small datasets or when the data is unsorted, as it doesn't require sorting or additional structure.
    * **Binary Search**: Best for large, sorted datasets, as it performs much more efficiently with a time complexity of O(log n).

# Screen Shots

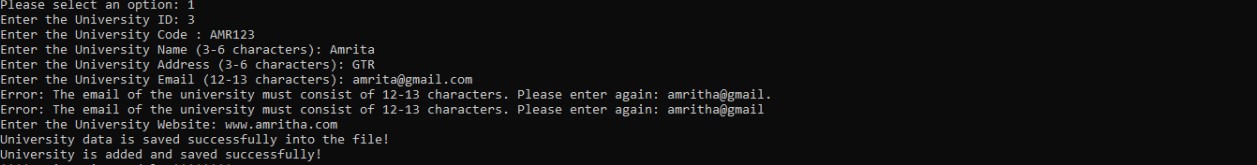
Retrieve University Details



Deleting a selected university



Creating a new university



Updating an existing university



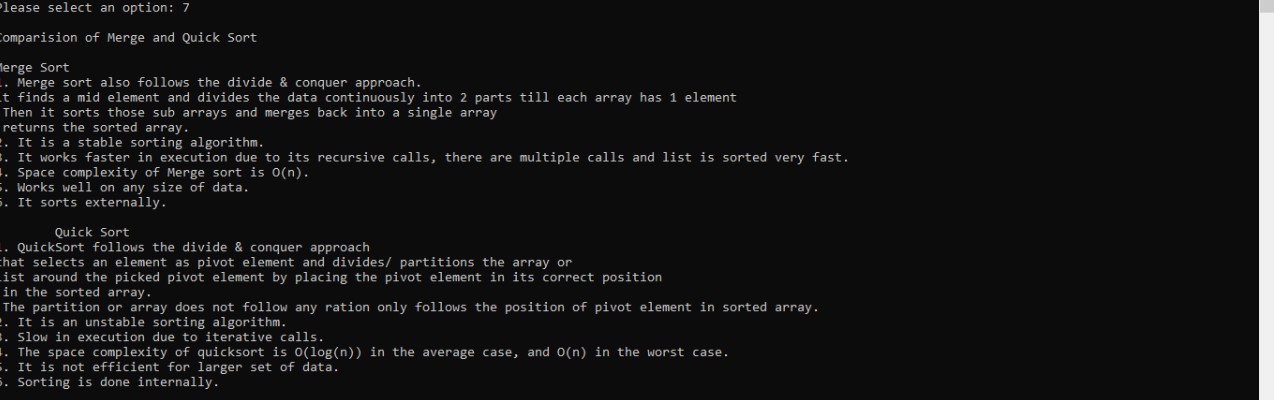
Sorting University Details



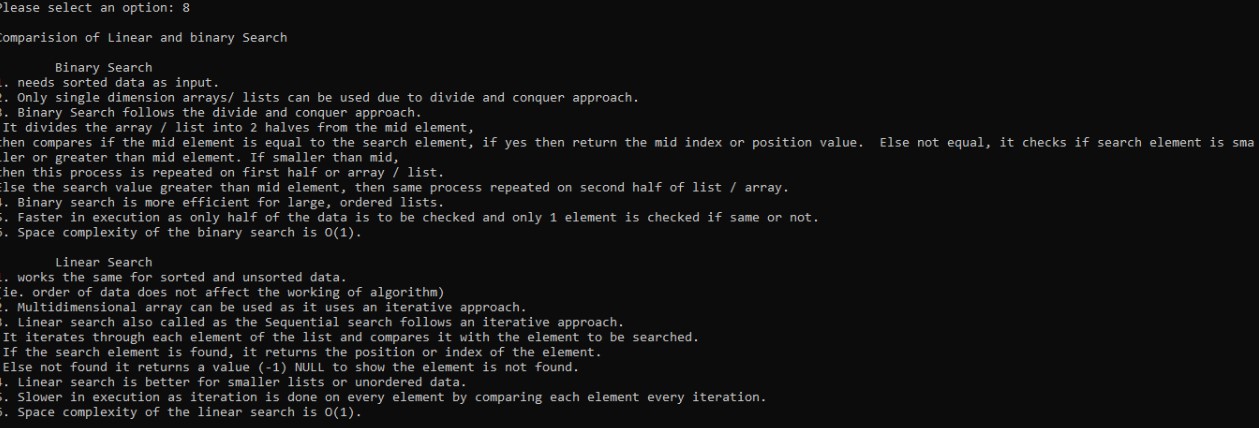
Searching University Details



Comparison of sorting algorithms



Comparison of searching algorithms



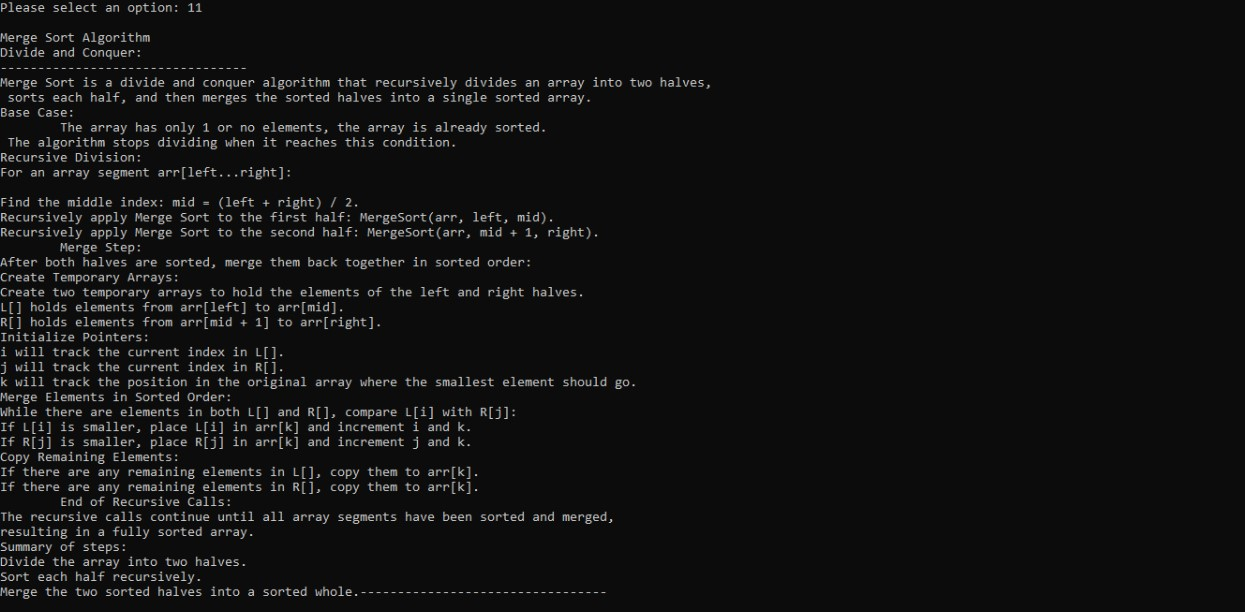
Time Complexity of sort algorithms

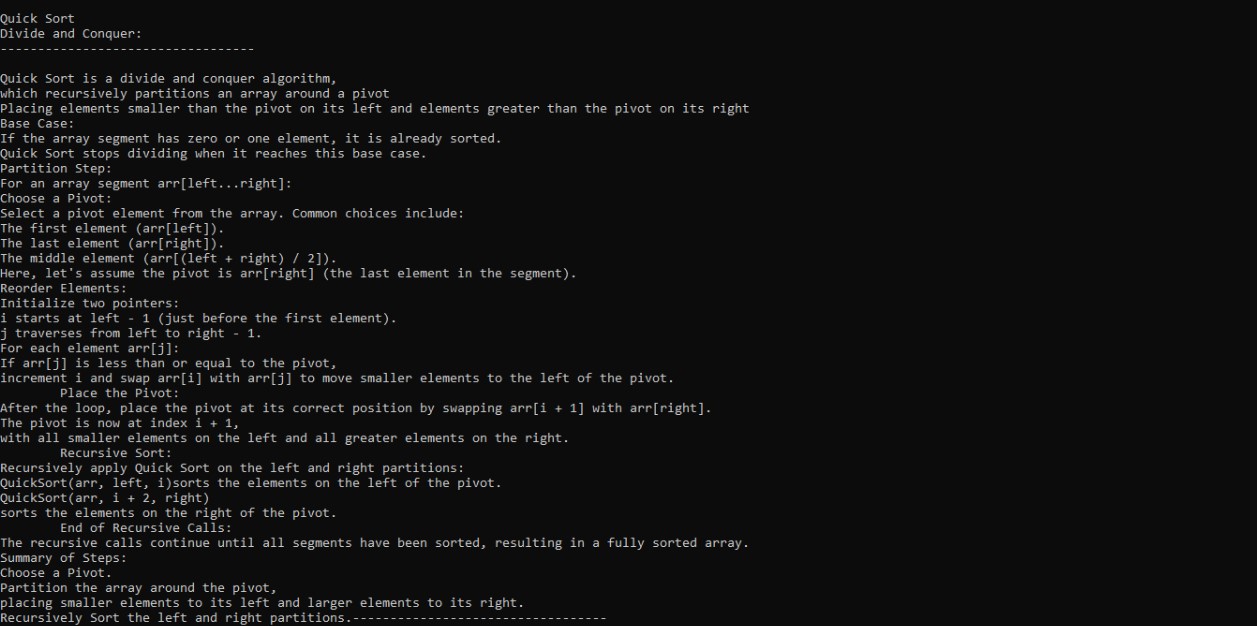


Time Complexity of Searching Algorithms



Sorting Algorithm details





Searching Algorithm details





# Conclusion:

This module effectively manages CRUD operations utilizing Merge Sort for sorting and Linear Search for searching. Merge Sort having time complexity of O(n log n) provides stable and efficient sorting,making it suitable for large data sets. For searching, Linear Search having time complexity of O(n) is used as it performs well on both dynamic and unsorted data set. The program also employes functions to display comparsion pseudocode and time complexities of searching and sorting algorithms. The combination of Merge Sort and Linear Search in this module balances simplicity with efficient data handling, making it versatile for moderate data sets.