**Department of Computer science and Engineering**

**CS 204: Design and Analysis of Algorithms**

**Project Title: OBE Implementation**

**Team Details:**

**Team Name: DREAM CODERS**

**Team project: Program School**

**Team members:**

**P. SRINIDHI AP23110011027**

**I. VINUTHNA AP23110011044**

**G. VENKAT AP23110011064**

**S. VAMSI AP23110011076**

**D. HARSHA VARDHAN AP23110011026**

**P.V.N.K. SREEHITHA AP23110011037**

**Designed by**

**GAVASKAR S,**

*Assistant Professor(Ad),*

*Department of CSE,*

*SRM University - AP.*

**INDEX**

Introduction

Modules in the project:

Architecture Diagram

Instructions:

Module Description

Programming Details naming conventions to be used:

Field/table details: Program Schools

Algorithm Details:

(i)Sorting

(ii)Searching

(ii) Storing the details in a text file 6

Sample coding Template

OBE\_MAIN\_DREAMCODERS.c

Instruction to use Sample Code:

ChatGPT Usage

Program Generated by ChatGPT

Instruction to use ChatGPT or other LLM Models

Introduction

# This C++ program is a student management system that uses a linked list to store and manage student records, including IDs, names, subjects, and marks. It allows users to add, update, remove students, and generate detailed reports for individual students or the entire list. The system includes features for searching student records by ID and sorting students based on their IDs or other criteria. Additionally, it supports saving and retrieving data from files, ensuring persistence across sessions. Users can set and display school details, such as the school name and location. The program operates through an interactive menu, providing a user-friendly way to manage student data, update marks, and generate reports, while also offering efficient searching and sorting capabilities for quick access to student information.

Project Module:

Program Schools

The primary module in this project is program management, focused on handling records for different programs.

This module allows:

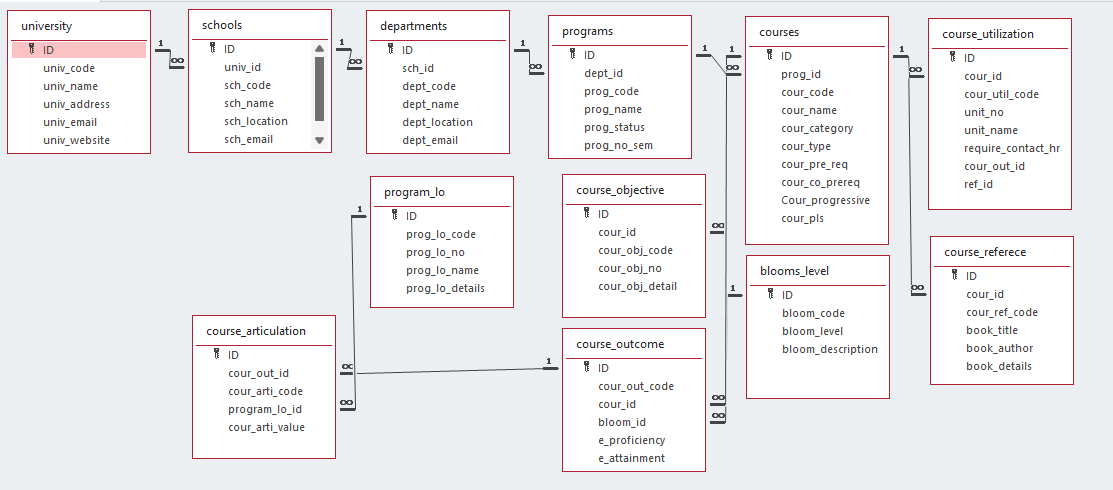
ID-based identification: Each program has a unique integer ID.

Attributed-based searching and sorting: Programs can be located and organized by their code, name for easier access and management.

CURD Operations: Users can create, retrieve, update, and delete program and records efficiently.

The Program Management module leverages Bubbles sort for sorting and Linear Search for basic searching, providing a straight forward approach to organizing and locating records.

# Architecture Diagram

****

## Instructions:

1. You have to consider each table name as a module name.
2. You have to also consider each table name as Structure name in C
3. You have to consider the fields in the table name as member variable of the structure
4. The data type of the field name ending with “\_id” is to be considered as integer and the rest of the field name is to be considered as char/string variable.
5. Ignore the relationship between the tables mentioned in the architecture diagram

# Module Description

**Module Name: PROGRAM MANAGEMENT**

**Module Description:**

This module is used to create, Update, Retrieve, Delete (known as CURD) details of the module and storing the details in the text file.

## Programming Details naming conventions to be used:

* **File name:** **Function/method name**
* **Function/method name**
  + **Create:** The\_Dreamcoders\_create\_program
  + **Update: The\_Dreamcoders\_update\_program**
  + **Retrieve: The\_Dreamcoders\_retrieve\_programs**
  + **Delete: The\_Dreamcoders\_retrieve\_programs**
  + **Sorting: The\_Dreamcoders\_sort\_by\_field**
  + **Searching: The\_Dreamcoders\_search\_by\_field**
  + **Storing: The\_Dreamcoders\_search\_by\_field**
  + **Comparison (both searching and Sorting)**:
    - For Searching - The\_Dreamcoders\_compare\_searching\_algorithms
    - For Sorting - The\_Dreamcoders\_compare\_sorting\_algorithms
  + **Time Complexity (both searching and Sorting):**
    - For Searching/sorting - The\_Dreamcoders\_display\_time\_complexity

* + **Algorithm Details (pseudocode or steps) (both searching and Sorting):**
    - For Searching/sorting - The\_Dreamcoders \_display\_time\_complexity
* **File name (for storing the details)**
  + File name to be used is :- schools\_setting .txt.

Field/table details: for programs

## 

|  |  |
| --- | --- |
| **Field Name** | **Data type** |
| id | Char |
| School\_code | Char |
| School\_name | char |
| School\_location | char |

## 1.Algorithm Details:

* **Setting School Details**
* **Input**: name and location of the school.
* **Process**:
  + Assign the provided name and location values to the schoolName and school Location member variables.
* **Output**:
  + Confirmation message displaying that the school details were successfully updated.

**2. Adding a Student**

* **Input**: Student object containing id, name, and a map of subjects with marks.
* **Process**:
  + Create a new Node with the Student data.
  + Set the next pointer of this new node to point to the current head of the linked list.
  + Update head to point to this new node.
* **Output**: Confirmation message that the student was successfully added.

**3. Removing a Student**

* **Input**: student ld (the ID of the student to remove).
* **Process**:
  + Traverse the linked list from the head:
* If a node with the matching student Id is found:
* Update the next pointer of the previous node to skip the node containing the student Id.
* If the node to be removed is the head, simply update head to the next node.
* Free the memory of the node.
* Return a success message.
* If the end of the list is reached without finding the student Id, return a "student not found" message.
* **Output**: Success message if removed or an error message if not found.

**4. Printing a Single Student Report**

* **Input**: student Id (the ID of the student whose report is requested).
* **Process**:
  + Traverse the linked list to locate the node with the matching student Id.
  + If found, print the student’s ID, name, and subject marks.
  + If not found, print an error message.
* **Output**: Student report or an error message.
* **Input**: None.

**Process**:

* + Traverse the entire linked list from head.
  + For each node, print the student’s ID, name, and their subject marks.
  + If the list is empty, print a message indicating no students to display.
* **Output**: Reports for all students or a message indicating an empty Delete each node to free allocated memory.

**5. Printing All Student Reports**

* **Input**: None.
* **Process**:
* Traverse the entire linked list from head.
* For each node, print the student’s ID, name, and their subject marks.
* If the list is empty, print a message indicating no students to display.
* **Output**: Reports for all students or a message indicating an empty list.

**6. Saving to a File**

* **Input**: filename (name of the file to save data to).
* **Process**:
  + Open the file in binary write mode.
  + Write the length of schoolName and schoolLocation and their contents to the file.
  + Traverse each node in the linked list:
    - Write the student's id and name.
    - For each subject, write the subject name and the mark.
  + Close the file.
* **Output**: Confirmation message indicating data was saved.

**7. Retrieving from a File**

* **Input**: filename (name of the file to load data from).
* **Process**:
  + Open the file in binary read mode.
  + Read the length and content of schoolName and schoolLocation.
  + Clear the existing linked list.
  + Read each student’s data, create a new Node for each, and add it to the linked list.
  + Close the file.
* **Output**: Confirmation message indicating data was loaded successfully.

**8. Updating Marks for a Specific Subject**

* **Input**: student Id (ID of the student), subject name, and newMark.
* **Process**:
  + Locate the node with the specified student Id.
  + If found, update the subject's mark with newMark in the subjects map.
  + If not found, print an error message.
* **Output**: Success message if updated or an error message if student not found.

**9. Finding a Student by ID**

* **Input**: student Id (ID of the student to locate).
* **Process**:
  + Traverse the linked list.
  + Return a pointer to the Student object if a node with the matching student Id is found.
  + If not found, return nullptr.
* **Output**: Pointer to Student object or nullptr if not found.

**10. Destructor for LinkedList**

* **Process**:
  + Traverse each node in the linked list starting from head.
  + Delete each node to free allocated memory.
* **Output**:
* No output; ensures memory cleanup upon program exit.

## 

## university.cs

*Console.WriteLine("\*\*\*\* OBE Application - Univeristy Setting \*\*\*\*\*\*\*\*");*

Source code

#include <iostream>

#include <vector>

#include <string>

#include <iomanip>

using namespace std;

struct Student {

string name;

string dob; // Date of Birth

string birthplace;

string motherName;

string fatherName;

string currentClass;

float annualFee;

float totalPaid;

float totalDue;

};

vector<Student> students;

void addStudent() {

Student newStudent;

cout << "Enter Student Name: ";

cin.ignore();

getline(cin, newStudent.name);

cout << "Enter Date of Birth (DD/MM/YYYY): ";

getline(cin, newStudent.dob);

cout << "Enter Birthplace: ";

getline(cin, newStudent.birthplace);

cout << "Enter Mother's Name: ";

getline(cin, newStudent.motherName);

cout << "Enter Father's Name: ";

getline(cin, newStudent.fatherName);

cout << "Enter Current Class: ";

getline(cin, newStudent.currentClass);

cout << "Enter Annual Fee: ";

cin >> newStudent.annualFee;

cout << "Enter Total Amount Paid: ";

cin >> newStudent.totalPaid;

newStudent.totalDue = newStudent.annualFee - newStudent.totalPaid;

students.push\_back(newStudent);

cout << "Student added successfully!\n";

}

void displayStudent(const Student& student) {

cout << "\n--- Student Details ---\n";

cout << "Name: " << student.name << endl;

cout << "Date of Birth: " << student.dob << endl;

cout << "Birthplace: " << student.birthplace << endl;

cout << "Mother's Name: " << student.motherName << endl;

cout << "Father's Name: " << student.fatherName << endl;

cout << "Current Class: " << student.currentClass << endl;

cout << "Annual Fee: " << fixed << setprecision(2) << student.annualFee << endl;

cout << "Total Amount Paid: " << fixed << setprecision(2) << student.totalPaid << endl;

cout << "Total Due: " << fixed << setprecision(2) << student.totalDue << endl;

}

void bubbleSort() {

for (size\_t i = 0; i < students.size() - 1; i++) {

for (size\_t j = 0; j < students.size() - i - 1; j++) {

if (students[j].name > students[j + 1].name) {

swap(students[j], students[j + 1]);

}

}

}

cout << "Students sorted by name.\n";

}

int binarySearch(const string& name) {

int left = 0;

int right = students.size() - 1;

while (left <= right) {

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

if (students[mid].name == name) {

return mid;

} else if (students[mid].name < name) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1; // Not found

}

void searchStudent() {

string name;

cout << "Enter Student Name to Search: ";

cin.ignore();

getline(cin, name);

int index = binarySearch(name);

if (index != -1) {

displayStudent(students[index]);

} else {

cout << "Student not found!\n";

}

}

void displayMenu() {

cout << "\nSchool Management System Menu:\n";

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

cout << "2. Search Student\n";

cout << "3. Sort Students by Name\n";

cout << "4. Exit\n";

}

int main() {

int choice;

while (true) {

displayMenu();

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

addStudent();

break;

case 2:

searchStudent();

break;

case 3:

bubbleSort();

break;

case 4:

cout << "Exiting the program. Goodbye!\n";

return 0;

default:

cout << "Invalid choice. Please try again.\n";

}

}

}

**Comparison of Sorting Algorithms**

BUBBLE SORT

#include <iostream>

using namespace std;

void bubbleSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++)

for (int j = 0; j < n - i - 1; j++)

if (arr[j] > arr[j + 1])

swap(arr[j], arr[j + 1]);

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr) / sizeof(arr[0]);

bubbleSort(arr, n);

for (int i = 0; i < n; i++) cout << arr[i] << " ";

return 0;

}

MERGE SORT

#include <iostream>

using namespace std;

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1, n2 = r - m;

int L[n1], R[n2];

for (int i = 0; i < n1; i++) L[i] = arr[l + i];

for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];

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

while (i < n1 && j < n2)

arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

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

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

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

int main() {

int arr[] = {38, 27, 43, 3, 9, 82, 10};

int n = sizeof(arr) / sizeof(arr[0]);

mergeSort(arr, 0, n - 1);

for (int i = 0; i < n; i++) cout << arr[i] << " ";

return 0;

}

Difference Between Bubble Sort and Merge Sort

1. Algorithm Type:

Bubble Sort: A simple comparison-based sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It is a quadratic sorting algorithm.

Merge Sort: A divide-and-conquer sorting algorithm that splits the array into halves, sorts them recursively, and merges the sorted halves. It is a divide-and-conquer sorting algorithm.

2. Time Complexity:

Bubble Sort:

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

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

Merge Sort:

Best, Average, and Worst Case: O(n log n).

3. Space Complexity:

Bubble Sort: O(1) (in-place sorting, no extra memory needed).

Merge Sort: O(n) (requires temporary arrays for merging).

4. Stability:

Both Bubble Sort and Merge Sort are stable algorithms, meaning they maintain the relative order of equal elements.

5. Use Cases:

Bubble Sort: Suitable for small datasets or educational purposes due to simplicity.

Merge Sort: Ideal for large datasets where efficiency is critical.

6. Recursion:

Bubble Sort: Non-recursive, iterative algorithm.

Merge Sort: Recursive algorithm.

7. Ease of Implementation:

Bubble Sort: Very simple to implement, beginner-friendly.

Merge Sort: Slightly more complex due to recursion and merging.

8. Performance:

Bubble Sort: Inefficient for large datasets due to its quadratic time complexity.

Merge Sort: Efficient for large datasets with its logarithmic time complexity.

Key Takeaway:

Use Bubble Sort for small, simple sorting tasks or when learning basic algorithms.

Use Merge Sort for larger datasets or when performance is critical.

**Comparison of Searching Algorithms**

1.BINARY SEARCH

#include <iostream>

using namespace std;

int binarySearch(int arr[], int n, int x) {

int l = 0, r = n - 1;

while (l <= r) {

int mid = l + (r - l) / 2;

if (arr[mid] == x) return mid;

if (arr[mid] < x) l = mid + 1;

else r = mid - 1;

}

return -1;

}

int main() {

int arr[] = {1, 3, 5, 7, 9};

int n = sizeof(arr) / sizeof(arr[0]);

int x;

cin >> x;

int result = binarySearch(arr, n, x);

cout << (result == -1 ? "Not Found" : "Found at index " + to\_string(result)) << endl;

return 0;

}

2.LINEAR SEARCH

#include <iostream>

using namespace std;

int linearSearch(int arr[], int n, int x) {

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

if (arr[i] == x) return i;

return -1;

}

int main() {

int arr[] = {10, 20, 30, 40, 50};

int n = sizeof(arr) / sizeof(arr[0]);

int x;

cin >> x;

int result = linearSearch(arr, n, x);

cout << (result == -1 ? "Not Found" : "Found at index " + to\_string(result)) << endl;

return 0;

}

Comparison Search Between Binary Search and Linear

1. Data Requirement:

Linear search works on both sorted and unsorted arrays, while binary search requires the array to be sorted before performing the search.

2. Time Complexity:

Linear search has a time complexity of O(n) because it checks every element one by one until it finds the target or reaches the end of the array. Binary search is more efficient with a time complexity of O(log n) as it divides the search range into halves repeatedly.

3. Space Complexity:

Both iterative binary search and linear search have a space complexity of O(1). However, recursive binary search has a space complexity of O(log n) due to the recursive call stack.

4. Best Case Performance:

In linear search, the best case is when the element is at the beginning of the array, taking only O(1) time.

In binary search, the best case occurs when the target is at the middle index, also taking O(1) time.

5. Worst Case Performance:

Linear search's worst case happens when the element is not in the array or at the very end, taking O(n) time.

Binary search's worst case occurs when the target is not found after repeatedly halving the array, taking O(log n) time.

6. Search Mechanism:

Linear search checks every element sequentially, making it straightforward but inefficient for large datasets. Binary search splits the search range in half, drastically reducing the number of comparisons.

7. Ease of Implementation:

Linear search is simple to implement as it only requires a loop. Binary search is slightly more complex due to the need for calculating the mid-point and handling conditions for sorted data.

8. Use Cases:

Linear search is suitable for small datasets or when the data isn't sorted.

Binary search is ideal for large, sorted datasets and scenarios where multiple searches are needed.

Outputs screen shots

A computer screen shot of a black screen

Description automatically generated

A computer screen shot of a black screen

Description automatically generated

A computer screen with a black screen

Description automatically generated

A computer screen shot of a black screen

Description automatically generated

A computer screen shot of a black screen

Description automatically generated

A computer screen with a black screen

Description automatically generated

**CONCLUSION**

This project creates a simple and effective system to manage bloom level data, allowing users to store, view, update, and delete information as needed. To keep data organized and easy to find, the program includes sorting and searching methods. Merge Sort is used for its steady and reliable performance, while Quick Sort provides faster sorting for unsorted data. For searching, Binary Search is used for quick lookups on sorted data, and Linear Search allows for flexible searches on unsorted data.

Overall, the project efficiently handles data operations and keeps information saved in a file, making sure that changes are kept even after the program closes. This project demonstrates basic principles of data management, showing how to choose and use different algorithms based on specific needs to create a reliable system for managing bloom level data.

THANK YOU!