**22BCP469D – Aryan Randeriya**

**20CP209P – Design & Analysis of Algorithms Lab**

**Practical 1**

1. **Insertion Sort**

**Implementation:**

#include <stdio.h>

void swap(int \**x*, int \**y*)

{

    int temp;

    temp = \**x*;

    \**x* = \**y*;

    \**y* = temp;

}

int main()

{

    int n; *// The number of elements*

    int arr[100], j, pivot;

    printf("Enter the number of elements: ");

    scanf("%d", &n);

*// Loop for user input for elements of the array*

    for (int i = 0; i < n; i++)

    {

        printf("Enter element at index %d: ", i);

        scanf("%d", &arr[i]);

    }

*//   Insertion Sort*

    for (int i = 1; i < n; i++)

    {

        pivot = arr[i]; *// Set pivot as the 2nd element and consider the first element sorted*

        j = i - 1;

        while (j >= 0 && arr[j] > pivot) *// loop from j=i-1 until an element larger than the pivot is found*

        {

            arr[j + 1] = arr[j]; *// Next element which is the larger (j+1) is assigned the value of the previous*

            j = j - 1; *// Decrement the index*

        }

        arr[j + 1] = pivot; *// The*

    }

    printf("Elements of the array: ");

    for (int i = 0; i < n; i++)

    {

        printf("%d ", arr[i]);

    }

    return 0;

}

**Output:**

A black screen with white text

Description automatically generated

**Time Complexity Analysis:**

* **Best Case –**
  + **O(n)**
* In the best-case scenario, the array is sorted, and the outer loop will always run n-1 times regardless if any shifting is doneor not. Everytime we try to enter the inner loop the condition of arr[j] > pivot is always false, meaning the pivot is always smaller.
* Total comparison operations = n-1 = O(n)
* Total shifting operations = 0 = O(1)
* **Total Best Case Time Complexity = O(n) + O(1) = O(n)**
* **Worst Case –**
  + **O(n2)**
* In the worst-case scenario, the array is reversely sorted, and the outer loop will run n-1 times and at each iteration, the inner loop will run I times because arr[j]> temp will be always true. To insert ith value at the correct position, we shift all the values from j=i-1 to 0 to one
* So at the ith iteration of the outer loop: Count of comparison operations=i and count of shifting operation=i.
* Total comparison operations =1+2+3+...+n-2+n-1=n(n-1)/2=O(n2).
* Total shifting operations = 1+2+3+...+n-2+n-1=n(n-1)/2=O(n2).
* **Total Worst Case Time Complexity = O(n2)+O(n2)=O(n2)**
* **Average Case –**
  + **O(n2)**
* When the input array is randomly ordered and each element must be moved to its correct position by shifting all the larger elements to the right. So, the average case time complexity is considered same as the worst case time complexity that is O(n^2)

**Selection Sort**

**Implementation:**

#include <stdio.h>

#include <conio.h>

void swap(int \**x*, int \**y*)

{

    int temp;

    temp = \**x*;

    \**x* = \**y*;

    \**y* = temp;

}

int main()

{

    int n; *// The number of elements*

    int arr[100], min;

    printf("Enter the number of elements: ");

    scanf("%d", &n);

*// Loop for user input for elements of the array*

    for (int i = 0; i < n; i++)

    {

        printf("Enter element at index %d: ", i);

        scanf("%d", &arr[i]);

    }

    for (int i = 0; i < n; i++) *// Iterate over all the elements once to set the minimum value*

    {

        min = i; *// Set the first element as the minimum*

        for (int j = i; j < n; j++) *// Loop until j = i -> n*

        {

            if (arr[j] < arr[min]) *// Compare if any element is smaller than the minimum*

            {

                swap(&arr[j], &arr[min]); *// If any element is smaller than minimum then swap them*

            }

        }

    }

    printf("Elements of the array: ");

    for (int i = 0; i < n; i++)

    {

        printf("%d ", arr[i]);

    }

    return 0;

}

**Output:**

A black screen with white text

Description automatically generated

**Time Complexity Analysis:**

* **Best Case –**
  + **O(n2)**
* When the input is already sorted, the comparison arr[j]<arr[min] becomes false every time, and the value of min will not get updated. Therefore, the sorted array is the best-case input for the selection sort.
* Total comparison operations = Count of nested loop iterations = O(n^2)
* The total count of swap operations = O(n).
* The total update operation of minindex = O(1).
* The time complexity of selection sort in the best case = O(n^2) +
* O(n) + O(1) = O(n^2).
* **Total Best Case Time Complexity = O(n^2) + O(n) + O(1) = O(n)**
* **Worst Case –**
  + **O(n2)**
* When the input is sorted in decreasing order, the comparison arr[j]<arr[min] becomes true every time, and the value of min will get updated every time. So the reverse-sorted array is the worst-case input for selection sort.
* The count of comparison operations = Total count of nested loop iterations = O(n^2).
* We perform one swap on each iteration of the outer loop. The total count of swapping operations = n\* O(1) =O(n).
* The total update operation of min= Total count of loop operations = O(n^2).
* **Total Worst Case Time Complexity O(n^2) +O(n) + O(n^2) = O(n^2).**
* **Average Case –**
  + **O(n2)**
* As both best case and worst-case complexity analysis are the same then the average case complexity will also be the same that is O(n^2)

**22BCP469D – Aryan Randeriya**

**20CP209P – Design & Analysis of Algorithms Lab**

**Practical 2**

1. **Merge Sort**

**Implementation:**

#include <stdio.h>

#include <stdlib.h>

*// Function to merge two subarrays of arr[]*

*// First subarray is arr[l..m]*

*// Second subarray is arr[m+1..r]*

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

{

    int n1 = *m* - *l* + 1;

    int n2 = *r* - *m*;

    int L[n1], R[n2];

*// Copy data to temporary arrays L[] and R[]*

    for (int i = 0; i < n1; i++)

        L[i] = *arr*[*l* + i];

    for (int j = 0; j < n2; j++)

        R[j] = *arr*[*m* + 1 + j];

    int i = 0, j = 0, k = *l*;

*// Merge the temporary arrays back into arr[l..r]*

    while (i < n1 && j < n2)

    {

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

    }

*// Copy the remaining elements from L[] if any*

    while (i < n1)

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

*// Copy the remaining elements from R[] if any*

    while (j < n2)

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

}

*// Main function that sorts arr[l..r] using merge()*

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

{

    if (*l* < *r*)

    {

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

*// Sort first and second halves*

        mergeSort(*arr*, *l*, m);

        mergeSort(*arr*, m + 1, *r*);

*// Merge the sorted halves*

        merge(*arr*, *l*, m, *r*);

    }

}

*// Function to print an array*

void printArray(int *A*[], int *size*)

{

    int i;

    for (i = 0; i < *size*; i++)

    {

        printf("%d ", *A*[i]);

    }

    printf("\n");

}

int main()

{

    int arr\_size;

    int arr[100];

    int i;

*// Input array size from user*

    printf("Enter array size : ");

    scanf("%d", &arr\_size);

*// Input array elements from user*

    for (i = 0; i < arr\_size; i++)

    {

        printf("Enter Element %d: ", i + 1);

        scanf("%d", &arr[i]);

    }

    printf("Given array is \n");

    printArray(arr, arr\_size);

*// Sort the array using merge sort*

    mergeSort(arr, 0, arr\_size - 1);

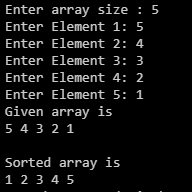
    printf("\nSorted array is \n");

    printArray(arr, arr\_size);

    return 0;

}

**Output:**



1. **Quick Sort**

**Implementation**

#include <stdio.h>

*// Function to swap two elements*

void swap(int \**a*, int \**b*)

{

    int t = \**a*;

    \**a* = \**b*;

    \**b* = t;

}

*// Function to partition the array and return the pivot index*

int partition(int *a*[], int *low*, int *high*)

{

    int pivot = *a*[*high*];

    int i = (*low* - 1);

    for (int j = *low*; j < *high*; j++)

    {

        if (*a*[j] <= pivot)

        {

            i++;

            swap(&*a*[i], &*a*[j]);

        }

    }

    swap(&*a*[i + 1], &*a*[*high*]);

    return (i + 1);

}

*// Function to perform quicksort recursively*

void quickSort(int *a*[], int *low*, int *high*)

{

    if (*low* < *high*)

    {

        int pi = partition(*a*, *low*, *high*);

        quickSort(*a*, *low*, pi - 1);

        quickSort(*a*, pi + 1, *high*);

    }

}

*// Function to print the array*

void printArray(int *a*[], int *size*)

{

    for (int i = 0; i < *size*; ++i)

    {

        printf("%d  ", *a*[i]);

    }

    printf("\n");

}

int main()

{

    int n, arr[100], i;

    printf("Enter array size: ");

    scanf("%d", &n);

    for (i = 0; i < arr\_size; i++)

    {

        printf("Enter Element %d: ", i + 1);

        scanf("%d", &arr[i]);

    }

    printf("Unsorted Array\n");

    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array in ascending order: \n");

    printArray(arr, n);

}

**Output**

A screenshot of a computer program

Description automatically generated

**22BCP469D – Aryan Randeriya**

**20CP209P – Design & Analysis of Algorithms Lab**

**Practical 3**

**Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers.**

**Implementation:**

#include <stdio.h>

#include <stdlib.h>

*// Node structure for a linked list*

struct Node

{

    int digit;

    struct Node \*link;

};

*// Function to print the linked list*

void printLinkedList(struct Node \**head*)

{

    int zerocounter = 0;

    struct Node \*current = *head*;

    while (current != NULL)

    {

        if (current->digit != 0) *// To Remove Trailing Zeroes*

            zerocounter = 1;

        if (!zerocounter == 0) *// Only Print the digits if the starting is not a zero*

            printf("%d", current->digit);

        current = current->link;

    }

    printf("\n");

}

*// Function to insert a node at the end of the linked list*

struct Node \*insertEnd(struct Node \**head*, int *digit*)

{

    struct Node \*newnode = (struct Node \*)malloc(sizeof(struct Node));

    newnode->digit = *digit*;

    newnode->link = NULL;

    if (*head* == NULL)

    {

        return newnode;

    }

    struct Node \*current = *head*;

    while (current->link != NULL)

    {

        current = current->link;

    }

    current->link = newnode;

    return *head*;

}

*// LinkedList Reverse Function*

struct Node \*reverse(struct Node \**head*)

{

    struct Node \*prev = NULL;

    struct Node \*current = *head*;

    struct Node \*next = NULL;

*// Reverse the links of each node in the linked list*

    while (current != NULL)

    {

        next = current->link;

        current->link = prev;

        prev = current;

        current = next;

    }

*// New head of the reversed list*

    return prev;

}

*// Function to insert a node at the begining of the linked list*

struct Node \*insertBeg(struct Node \**head*, int *digit*)

{

    struct Node \*newnode = (struct Node \*)malloc(sizeof(struct Node));

    newnode->digit = *digit*;

    newnode->link = *head*;

    return newnode;

}

*// Addition Function*

struct Node \*addition(struct Node \**number1*, struct Node \**number2*)

{

    int addition = 0;

    int carry = 0;

    struct Node \*start1 = *number1*;

    struct Node \*start2 = *number2*;

    struct Node \*result = NULL;

*// Traverse both linked lists until both are not null*

    while ((start1 != NULL) && (start2 != NULL))

    {

*// Calculate the sum (0 to 9 values only) and update 'addition'*

        addition = (start1->digit + start2->digit + carry) % 10;

*// Insert the sum into the result linked list*

        result = insertBeg(result, (addition));

*// Update the carry for the next iteration*

        carry = (start1->digit + start2->digit + carry) / 10;

        start1 = start1->link;

        start2 = start2->link;

    }

*// If there are nodes still left in start2*

    if (start1 == NULL)

    {

        while (start2 != NULL)

        {

            addition = (start2->digit + carry) % 10;

            result = insertBeg(result, addition);

            carry = (start2->digit + carry) / 10;

            start2 = start2->link;

        }

    }

*// If there are nodes still left in start1*

    if (start2 == NULL)

    {

        while (start1 != NULL)

        {

            addition = (start1->digit + carry) % 10;

            result = insertBeg(result, addition);

            carry = (start1->digit + carry) / 10;

            start1 = start1->link;

        }

    }

*// If there exists a carry after additioning the last digit, resolve it here*

    if ((start1 == NULL) && (start2 == NULL) && (carry != 0))

        result = insertBeg(result, carry);

    return result;

}

*// Multiplication Function*

void multiplication(struct Node \**number1*, struct Node \**number2*)

{

    int carry = 0;

    int count = 0;

    struct Node \*start1 = *number1*;

    struct Node \*start2 = *number2*;

    struct Node \*result = NULL;

    while (start2 != NULL)

    {

        struct Node \*temp = start1;

        struct Node \*answer = NULL;

*// Append zeros to the result for the current count*

        int x = count;

        while (x > 0)

        {

            answer = insertEnd(answer, 0);

            x--;

        }

        while (temp != NULL)

        {

            int multiply = (temp->digit \* start2->digit + carry) % 10;

            carry = (temp->digit \* start2->digit + carry) / 10;

            answer = insertBeg(answer, multiply);

            temp = temp->link;

        }

*// Carry handling if generated*

        if (carry > 0)

        {

            answer = insertBeg(answer, carry);

            carry = 0;

        }

*// Reverse the answer and add it to the result*

        result = addition(result, answer);

        start2 = start2->link;

        count++;

    }

    result = reverse(result);

    printf("\nResult of multiplication :\n");

    printLinkedList(result);

}

*// Subtraction Function*

void subtraction(struct Node \**number1*, struct Node \**number2*)

{

    int difference = 0, appendMinus = 0, borrow = 0;

    struct Node \*start1 = *number1*;

    struct Node \*start2 = *number2*;

    struct Node \*temp;

    struct Node \*result = NULL;

    if (*number1* < *number2*)

    {

*// Swap both numbers when first number is smaller*

        temp = start1;

        start1 = start2;

        start2 = temp;

        appendMinus = 1;

    }

    while (start1 != NULL || start2 != NULL)

    {

        difference = (start1->digit - borrow);

        if (start2 != NULL)

        {

            difference -= start2->digit;

            start2 = start2->link;

        }

        if (difference < 0)

        {

            difference += 10;

            borrow = 1;

        }

        else

            borrow = 0;

*// Insert the difference into the result linked list*

        result = insertBeg(result, difference);

        start1 = start1->link;

    }

*// If the first number is smaller than multiple by -1*

    if (appendMinus == 1)

        printf("\nResult of subtraction :\n -");

    else

        printf("\nResult of subtraction :\n");

    printLinkedList(result);

}

int main()

{

    struct Node \*start1 = NULL;

    struct Node \*start2 = NULL;

    struct Node \*result;

    int digit, operation, number1, number2;

    printf("Enter number 1: ");

    scanf("%d", &number1);

    printf("Enter number 2: ");

    scanf("%d", &number2);

    while (number1 != 0)

    {

*// Reverse the digits*

        digit = number1 % 10;

        number1 = number1 / 10;

        start1 = insertEnd(start1, digit); *// Insert the digit at the ending of the list*

    }

    while (number2 != 0)

    {

*// Reverse the digits*

        digit = number2 % 10;

        number2 = number2 / 10;

        start2 = insertEnd(start2, digit); *// Insert the digit at the ending of the list*

    }

    while (1)

    {

        printf("\n1. Addition\n2. Multiplication\n3. Subtraction\n4. Exit\n\n");

        scanf("%d", &operation);

        switch (operation)

        {

        case 1:

            result = addition(start1, start2);

            printf("\nResult of addition :\n");

            printLinkedList(result);

            break;

        case 2:

            multiplication(start1, start2);

            break;

        case 3:

            subtraction(start1, start2);

            break;

        case 4:

            printf("Exiting...");

            exit(0);

        default:

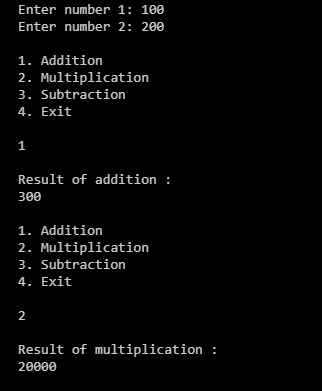
            printf("Please choose a valid option.\n");

            break;

        }

    }

}

**Output:**

A screenshot of a computer program

Description automatically generated

**22BCP469D – Aryan Randeriya**

**20CP209P – Design & Analysis of Algorithms Lab**

**Practical 4**

1. **Linked List**

**Implementation:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

struct Record

{

    char name[500];

    int x;

    int y;

    struct Record \*link;

};

struct Record \*start = NULL;

struct Record \*createNewNode(char *name*[], int *x*, int *y*)

{

    struct Record \*newNode = (struct Record \*)malloc(sizeof(struct Record));

    strncpy(newNode->name, *name*, sizeof(newNode->name));

    newNode->x = *x*;

    newNode->y = *y*;

    newNode->link = NULL;

    printf("Created new Node with name of %s\n", *name*);

    return newNode;

}

void insert()

{

    int x, y;

    char name[500];

    printf("Inserting Data...\n");

    printf("Enter Name of city: ");

    scanf("%s", name);

    printf("Enter X coordinates of city: ");

    scanf("%d", &x);

    printf("Enter Y coordinates of city: ");

    scanf("%d", &y);

    struct Record \*newNode = createNewNode(name, x, y);

    if (start == NULL)

    {

        start = newNode;

        return;

    }

    else

    {

        newNode->link = start;

        start = newNode;

    }

}

void display()

{

    if (start == NULL)

    {

        printf("There are no Records!\n");

        return;

    }

    int counter;

    struct Record \*temp = start;

    while (temp != NULL)

    {

        counter++;

        printf(" %d : [City - %s : ", counter, &temp->name);

        printf("X - %d : ", temp->x);

        printf("Y - %d] \n", temp->y);

        temp = temp->link;

    }

}

void deleteByName()

{

    struct Record \*previous = NULL;

    struct Record \*current = start;

    char name[500];

    printf("Enter Name of City to delete record:");

    scanf("%s", &name);

    while (current != NULL && strcmp(current->name, name) != 0)

    {

        previous = current;

        current = current->link;

    }

    if (current == NULL)

    {

        printf("No such Record Exists!\n");

        return;

    }

    if (previous == NULL) *// Current element is the first element in the list*

    {

        start = current->link;

    }

    else *// If the current element is not the first element of the list*

    {

        previous->link = current->link;

    }

    free(current);

    printf("Record has been deleted!\n");

    display();

}

void deleteByCoordinates()

{

    struct Record \*previous = NULL;

    struct Record \*current = start;

    int x, y;

    printf("Enter X coordinate to delete record: ");

    scanf("%d", &x);

    printf("Enter Y coordinate to delete record: ");

    scanf("%d", &y);

    while (current != NULL && (current->x != x) && (current->y != y))

    {

        previous = current;

        current = current->link;

    }

    if (current == NULL)

    {

        printf("No such Record Exists!\n");

        return;

    }

    if (previous == NULL) *// Current element is the first element in the list*

    {

        start = current->link;

    }

    else *// If the current element is not the first element of the list*

    {

        previous->link = current->link;

    }

    free(current);

    printf("Record has been deleted!\n");

    display();

}

void searchByCoordinates()

{

    struct Record \*previous = NULL;

    struct Record \*current = start;

    int x, y;

    printf("Enter X coordinate to display record details: ");

    scanf("%d", &x);

    printf("Enter Y coordinate to display record details: ");

    scanf("%d", &y);

    while (current != NULL && (current->x != x) && (current->y != y))

    {

        previous = current;

        current = current->link;

    }

    if (current == NULL)

    {

        printf("No such Record Exists!\n");

        return;

    }

    printf("Record with Coordinates X: %d Y: %dFound!\n", current->x, current->y);

    printf("City Name: %s\n", current->name);

}

void searchByName()

{

    struct Record \*previous = NULL;

    struct Record \*current = start;

    char name[500];

    printf("Enter Name of City to display all details:");

    scanf("%s", &name);

    while (current != NULL && strcmp(current->name, name) != 0)

    {

        previous = current;

        current = current->link;

    }

    if (current == NULL)

    {

        printf("No such Record Exists!\n");

        return;

    }

    printf("Record %s Found!\n", current->name);

    printf("X: %d\n", current->x);

    printf("Y: %d\n", current->y);

}

void displayRecordsWithinDistance()

{

    int x0, y0, dist;

    struct Record \*temp = start;

    printf("Enter X coordinates of city: ");

    scanf("%d", &x0);

    printf("Enter Y coordinates of city: ");

    scanf("%d", &y0);

    printf("Enter distance from point: ");

    scanf("%d", &dist);

    printf("Records at given distance\n");

*// Traverse the list and calculate the distance of each record from the specified point*

    while (temp != NULL)

    {

        double distance = sqrt(pow(temp->x - x0, 2) + pow(temp->y - y0, 2));

*// If the distance is less than or equal to the given distance, print the record details*

        if (distance <= dist)

        {

            printf("[City = %s , X = %d , Y = %d , Distance = %lf]\n", temp->name, temp->x, temp->y, dist);

        }

        temp = temp->link;

    }

}

int main()

{

    int option;

    while (1)

    {

        printf("Enter an option: \n1. Insert a Record.\n2. Delete a record by name\n3. Delete a record by coordinate.\n4. Search a record by name\n5. Search a record by coordinate\n6. Print all records within a given distance of a specified point.\n7. Display All Records\n8. Exit\n\n");

        scanf("%d", &option);

        switch (option)

        {

        case 1:

            insert();

            break;

        case 2:

            deleteByName();

            break;

        case 3:

            deleteByCoordinates();

            break;

        case 4:

            searchByName();

            break;

        case 5:

            searchByCoordinates();

            break;

        case 6:

            displayRecordsWithinDistance();

            break;

        case 7:

            display();

            break;

        case 8:

            exit(0);

        default:

            break;

        }

    }

    return 0;

}

**Output:**

A computer screen shot of a black screen

Description automatically generatedA black screen with white text

Description automatically generatedA black background with white text

Description automatically generatedA black screen with white text

Description automatically generatedA black screen with white text

Description automatically generated A black background with white text

Description automatically generatedA black screen with white text

Description automatically generatedA black screen with white text

Description automatically generated

1. **Array**

**Implementation:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#define max\_size 1000 *// Maximum number of records*

struct Record

{

    char name[500];

    int x;

    int y;

};

struct Record database[max\_size]; *// Create an array of structures*

int size\_count = -1;

void display()

{

    printf("Size of Database: %d \n", size\_count + 1);

    if (size\_count == -1)

    {

        printf("There are no Records!\n");

        return;

    }

    for (int i = 0; i <= size\_count; i++)

    {

        printf("\n%d: [City - %s,", i + 1, database[i].name);

        printf("X - %d,", database[i].x);

        printf("Y - %d]\n", database[i].y);

    }

}

void insert()

{

    int x, y;

    char name[500];

    printf("Inserting Data...\n");

    printf("Enter Name of city: ");

    scanf("%s", name);

    printf("Enter X coordinates of city: ");

    scanf("%d", &x);

    printf("Enter Y coordinates of city: ");

    scanf("%d", &y);

    if (size\_count >= max\_size)

    {

        printf("Database at Full Capacity!\n");

        return;

    }

    size\_count++;

    strncpy(database[size\_count].name, name, sizeof(database[size\_count].name));

    database[size\_count].x = x;

    database[size\_count].y = y;

    printf("\nCity %s Succesfully Inserted!\n\n", name);

}

void delete(int *index*)

{

    size\_count--;

    for (int i = *index*; i <= size\_count; i++)

    {

        database[i] = database[i + 1];

    }

    printf("Successfully Deleted!\n");

    display();

}

void deleteByName()

{

    char name[500];

    int indextodelete;

    printf("Enter City Name of record to delete: ");

    scanf("%s", name);

    for (int i = 0; i <= size\_count; i++)

    {

        if (strcmp(name, database[i].name) == 0)

        {

            indextodelete = i;

            delete (indextodelete);

            return;

        }

    }

    printf("No Such Record Exists!\n");

}

void deleteByCoordinates()

{

    char name[500];

    int indextodelete;

    int x, y;

    printf("Enter X coordinate to delete record: ");

    scanf("%d", &x);

    printf("Enter Y coordinate to delete record: ");

    scanf("%d", &y);

    for (int i = 0; i <= size\_count; i++)

    {

        if (database[i].x == x && database[i].y == y)

        {

            indextodelete = i;

            delete (indextodelete);

            return;

        }

    }

    printf("No Such Record Exists!\n");

}

void searchByCoordinates()

{

    int x, y;

    printf("Enter X coordinate to display record details: ");

    scanf("%d", &x);

    printf("Enter Y coordinate to display record details: ");

    scanf("%d", &y);

    for (int i = 0; i <= size\_count; i++)

    {

        if (database[i].x == x && database[i].y == y)

        {

            printf("Record found!");

            printf("\n%d: [City - %s,", i + 1, database[i].name);

            printf("X - %d,", database[i].x);

            printf("Y - %d]\n", database[i].y);

            return;

        }

    }

    printf("No Such Record Exists!\n");

}

void searchByName()

{

    char name[500];

    printf("Enter a name to search");

    scanf("%s", name);

    for (int i = 0; i <= size\_count; i++)

    {

        if (strcmp(name, database[i].name) == 0)

        {

            printf("Record found!");

            printf("\n%d: [City - %s,", i + 1, database[i].name);

            printf("X - %d,", database[i].x);

            printf("Y - %d]\n", database[i].y);

            return;

        }

    }

    printf("No Such Record Exists!\n");

}

void displayRecordsWithinDistance()

{

    int x0, y0, dist;

    printf("Enter X coordinates of city: ");

    scanf("%d", &x0);

    printf("Enter Y coordinates of city: ");

    scanf("%d", &y0);

    printf("Enter distance from point: ");

    scanf("%d", &dist);

    printf("Records at given distance\n");

*// Traverse the list and calculate the distance of each record from the specified point*

    for (int i = 0; i <= size\_count; i++)

    {

        double distance = sqrt(pow(database[i].x - x0, 2) + pow(database[i].y - y0, 2));

*// If the distance is less than or equal to the given distance, print the record details*

        if (distance <= dist)

        {

            printf("[City = %s , X = %d , Y = %d , Distance = %lf]\n", database[i].name, database[i].x, database[i].y, dist);

        }

    }

}

int main()

{

    int option;

    while (1)

    {

        printf("Enter an option: \n1. Insert a Record.\n2. Delete a record by name\n3. Delete a record by coordinate.\n4. Search a record by name\n5. Search a record by coordinate\n6. Print all records within a given distance of a specified point.\n7. Display All Records\n8. Exit\n\n");

        scanf("%d", &option);

        switch (option)

        {

        case 1:

            insert();

            break;

        case 2:

            deleteByName();

            break;

        case 3:

            deleteByCoordinates();

            break;

        case 4:

            searchByName();

            break;

        case 5:

            searchByCoordinates();

            break;

        case 6:

            displayRecordsWithinDistance();

            break;

        case 7:

            display();

            break;

        case 8:

            exit(0);

        default:

            break;

        }

    }

    return 0;

}

**Output:**

A computer screen shot of a black screen

Description automatically generatedA black screen with white text

Description automatically generatedA black background with white text

Description automatically generatedA black screen with white text

Description automatically generatedA black screen with white text

Description automatically generated A black background with white text

Description automatically generatedA black screen with white text

Description automatically generatedA black screen with white text

Description automatically generated