**Lab no: 19** **Date:2081/12/28**

**Title: WAP to sort user input data using insertion sort in C**.

**Insertion Sort**

Insertion Sort is a simple and intuitive sorting algorithm. It works by building a sorted portion of the list one element at a time, by inserting each new element into its correct position within the sorted portion**.**

**Pseudocode**

void insertionSort(int A[], int n) {

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

key = A[i];

j = i - 1;

while (j >= 0 && A[j] > key) {

A[j + 1] = A[j];

j = j - 1;

}

A[j + 1] = key;

}

}

**IDE: VS Code**

**Language: C**

**Source Code**

#include <stdio.h>

void insertionsort(int arr[], int n, int order) {

    int i, key, j;

    for (i = 1; i < n; i++) {

        key = arr[i];

        j = i - 1;

        // ASCENDING order

        if (order == 1) {

            while (j >= 0 && arr[j] > key) {

                arr[j + 1] = arr[j];

                j = j - 1;

            }

        }

        // DESCENDING order

        else if (order == 2) {

            while (j >= 0 && arr[j] < key) {

                arr[j + 1] = arr[j];

                j = j - 1;

            }

        }

        arr[j + 1] = key;

        // Display current state of array

        printf("Step %d: ", i);

        for (int k = 0; k < n; k++) {

            printf("%d ", arr[k]);

        }

        printf("\n");

    }

}

int main() {

    int n, order;

    printf("\*\*\tCompiled By Aswin Phuyal\t\*\*\n");

    printf("Enter number of elements: ");

    scanf("%d", &n);

    int arr[n];

    printf("Enter elements:\n");

    for (int i = 0; i < n; i++) {

        scanf("%d", &arr[i]);

    }

    while (1) {

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

        printf("1. Ascending\n");

        printf("2. Descending\n");

        printf("0. Exit\n");

        printf("Enter choice (0, 1 or 2): ");

        scanf("%d", &order);

        if (order == 0) {

            printf("Exiting...\n");

            break;

        } else if (order != 1 && order != 2) {

            printf("Invalid choice. Try again.\n");

            continue;

        }

        printf("\nSorting steps:\n");

        insertionsort(arr, n, order);

        printf("\nSorted array: ");

        for (int i = 0; i < n; i++) {

            printf("%d ", arr[i]);

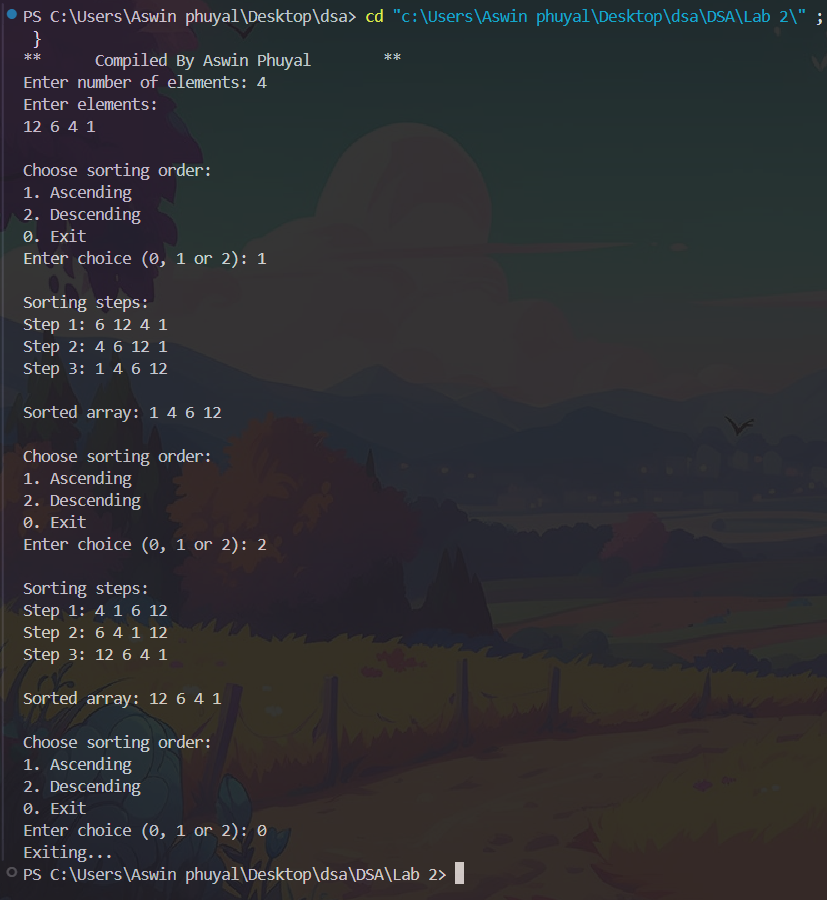
        }

        printf("\n");

    }

    return 0;

}

**Output:**

**Lab no: 17** **Date:2081/12/28**

**Title: WAP to sort user input data using bubble sort in C**.

**Bubble Sort**

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. The process continues until the entire list is sorted.

**Pseudocode:**

bubblesort(int arr[], int n) {

int i, j, temp;

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

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

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

temp = arr[j];

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

arr[j+1] = temp;

}

}

}

**IDE: VS Code**

**Language: C**

**Source Code:**

#include <stdio.h>

// bubble sort func and trace each pass

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

int i, j, temp;

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

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

// compare adjacent elements

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

// swap the elements if they are in the wrong order

temp = arr[j];

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

arr[j+1] = temp;

}

}

// display array after each pass

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

for (int k = 0; k < n; k++) {

printf("%d ", arr[k]);

}

printf("\n");

}

}

int main() {

int n;

printf("\*\*\tCompiled By Aswin phuyal\t\*\*\n");

printf("BUBBLE SORT\n");

// take input from the user

printf("enter no. of elements: ");

scanf("%d", &n);

int arr[n];

printf("enter elements:\n");

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

scanf("%d", &arr[i]);

}

// func call

printf("\nsorting process:\n");

bubblesort(arr, n);

// final sorted array

printf("\nsorted array: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

**Output:**



**Lab no: 11** **Date:2081/12/28**

**Title: WAP for BST Insertion, Deletion, Traversal and Search in C**.

**BST**

A Binary Search Tree (BST) is a type of binary tree where each node has at most two children (left and right). It is structured in such a way that the values of nodes on the left subtree of a node are less than the node's value, and the values of nodes on the right subtree are greater than the node's value. This property makes BSTs efficient for various operations such as searching, insertion, and deletion.

**Basic Operations on a BST:**

**Search**

To find a value, you start from the root and traverse left if the value is smaller than the current node, or right if it is larger. This process continues until the value is found or the search ends at a null pointer.

**Insertion**

To insert a new value, you begin at the root. If the new value is less than the current node, you move to the left child. If it is greater, you move to the right child. This continues until you find an empty spot (a null child) where the new value can be placed.

**Deletion**

Deletion is a bit more complex. If the node to be deleted has no children, you can simply remove it. If it has one child, you bypass the node and connect its parent to its only child. If the node has two children, you replace the node with its in-order successor or in-order predecessor

**Traversals**

**In-order traversal**

Traverse the left subtree, visit the root, and then traverse the right subtree. This traversal gives the values in ascending order.

**Pre-order traversal**

Visit the root first, then traverse the left subtree and the right subtree.

**Post-order traversal**

Traverse the left subtree, then the right subtree, and visit the root last.

**IDE: VS Code**

**Language: C**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node \*left, \*right;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

if (!newNode) {

printf("Memory allocation failed.\n");

return NULL;

}

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to insert a node in the BST

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL)

return createNode(data);

if (data < root->data)

root->left = insert(root->left, data);

else if (data > root->data)

root->right = insert(root->right, data);

return root;

}

// Function to search for a value

struct Node\* search(struct Node\* root, int key) {

if (root == NULL || root->data == key)

return root;

if (key < root->data)

return search(root->left, key);

else

return search(root->right, key);

}

// Find the minimum value node in the right subtree

struct Node\* findMin(struct Node\* node) {

struct Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

// Function to delete a node from the BST

struct Node\* deleteNode(struct Node\* root, int key) {

if (root == NULL)

return root;

if (key < root->data)

root->left = deleteNode(root->left, key);

else if (key > root->data)

root->right = deleteNode(root->right, key);

else {

// Node found

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct Node\* temp = root->left;

free(root);

return temp;

}

// Node with two children

struct Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

// In-order Traversal (Left -> Root -> Right)

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

// Pre-order Traversal (Root -> Left -> Right)

void preorder(struct Node\* root) {

if (root != NULL) {

printf("%d ", root->data);

preorder(root->left);

preorder(root->right);

}

}

// Post-order Traversal (Left -> Right -> Root)

void postorder(struct Node\* root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

printf("%d ", root->data);

}

}

int main() {

printf("\*\*\tCompiled By Aswin Phuyal\t\*\*\n");

struct Node\* root = NULL;

int choice, value;

while (1) {

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

printf("1. Insert\t2. Delete\t3. Search\n");

printf("4. In-order Traversal\t5. Pre-order Traversal\t6. Post-order Traversal\n");

printf("7. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert: ");

scanf("%d", &value);

root = insert(root, value);

printf("%d inserted successfully.\n", value);

break;

case 2:

printf("Enter value to delete: ");

scanf("%d", &value);

root = deleteNode(root, value);

printf("%d deleted successfully (if it existed).\n", value);

break;

case 3:

printf("Enter value to search: ");

scanf("%d", &value);

struct Node\* searchResult = search(root, value);

if (searchResult)

printf("%d found in the tree.\n", value);

else

printf("%d not found in the tree.\n", value);

break

case 4:

printf("In-order Traversal: ");

inorder(root);

printf("\n");

break;

case 5:

printf("Pre-order Traversal: ");

preorder(root);

printf("\n");

break;

case 6:

printf("Post-order Traversal: ");

postorder(root);

printf("\n");

break;

case 7:

printf("Exited.\n");

return 0;

default:

printf("Invalid choice. Try again.\n");

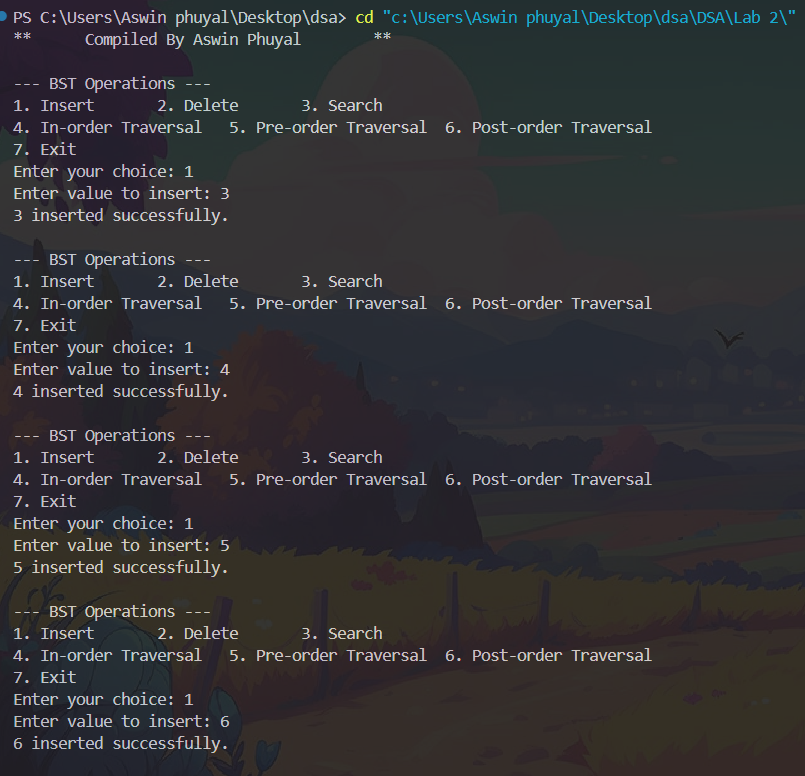
break;

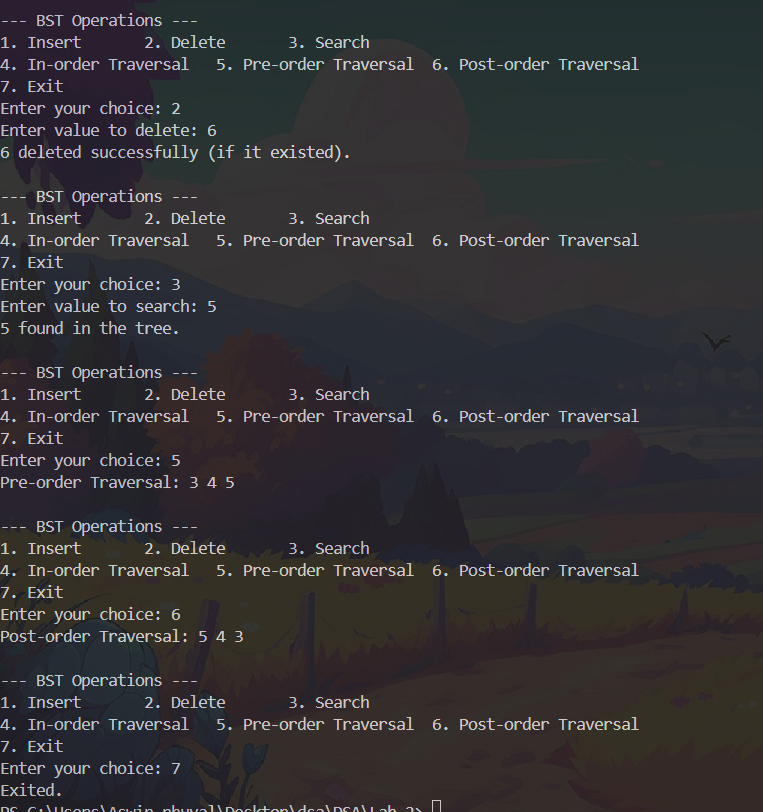
}

}

return 0;

}

**Output:**

****

**Lab no: 12** **Date:2081/12/28**

**Title: WAP to calculate shortest path using Dijkastra Algorithm in C**.

**Dijkastra Algorithm**

Dijkstra's Algorithm is a graph search algorithm used to find the shortest path from a source vertex to all other vertices in a weighted graph. The graph can be either directed or undirected.

**Pseudocode:**

void Dijkstra(int src, int node) {

int\* distance = (int\*)malloc((node + 1) \* sizeof(int));

bool\* visited = (bool\*)malloc((node + 1) \* sizeof(bool));

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

distance[i] = INT\_MAX;

visited[i] = false;

}

distance[src] = 0;

for (int count = 1; count <= node - 1; count++) {

int u = minDistance(distance, visited, node);

visited[u] = true;

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

if (!visited[i] && mat[u][i]) {

if (distance[i] > distance[u] + mat[u][i])

distance[i] = distance[u] + mat[u][i];

}

}

}

**IDE: VS Code**

**Language: C**

**Source Code:**

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#include <stdlib.h>

int mat[99][99] = {0};

void Edges(int numedge) {

int node1, node2, weight;

printf("Enter (node1) -> (node2) = weight:\n");

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

scanf("%d%d%d", &node1, &node2, &weight);

mat[node1][node2] = mat[node2][node1] = weight;

}

}

// Function to find the vertex with the minimum distance value

int minDistance(int\* distance, bool\* visited, int node) {

int min = INT\_MAX;

int min\_index = -1;

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

if (!visited[i]) {

if (distance[i] < min) {

min = distance[i];

min\_index = i;

}

}

}

return min\_index;

}

// Function to find the shortest path using Dijkstra's Algorithm

void Dijkstra(int src, int node) {

int\* distance = (int\*)malloc((node + 1) \* sizeof(int)); // Dynamic allocation

bool\* visited = (bool\*)malloc((node + 1) \* sizeof(bool)); // Dynamic allocation

// Initialize distance array and visited array

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

distance[i] = INT\_MAX;

visited[i] = false;

}

distance[src] = 0;

// Apply Dijkstra's algorithm

for (int count = 1; count <= node - 1; count++) {

int u = minDistance(distance, visited, node);

visited[u] = true;

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

if (!visited[i] && mat[u][i]) {

if (distance[i] > distance[u] + mat[u][i])

distance[i] = distance[u] + mat[u][i];

}

}

}

// Display the results

printf("Shortest path using Dijkstra's Algorithm :\n");

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

printf("%12s", "Node |");

for (int i = 1; i <= node; i++)

printf("%3d |", i);

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

printf("%12s", "Distance |");

for (int i = 1; i <= node; i++)

printf("%3d |", distance[i]);

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

// Free allocated memory

free(distance);

free(visited);

}

int main() {

int node, numedges, src;

printf("\*\*\tCompiled By Aswin phuyal\t\*\*\n");

printf("Enter number of nodes : ");

scanf("%d", &node);

printf("Enter number of edges : ");

scanf("%d", &numedges);

// Take input for edges

Edges(numedges);

printf("Enter the source node : ");

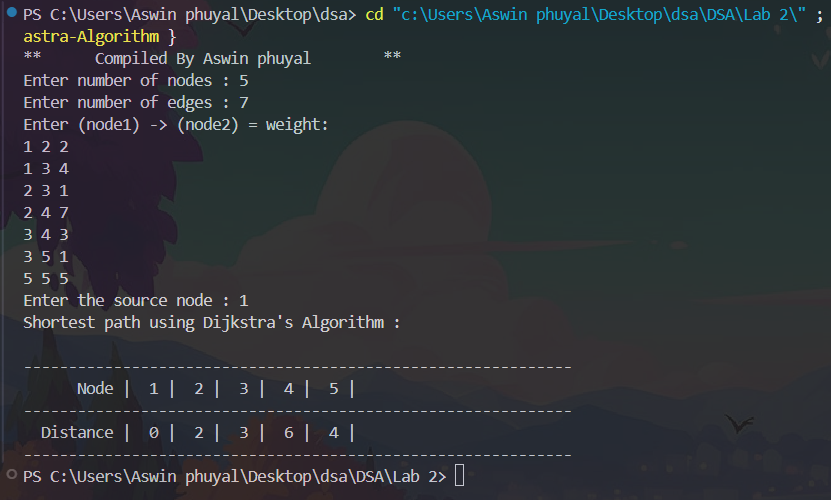
scanf("%d", &src);

// Find and display the shortest paths

Dijkstra(src, node);

return 0;

}

**Output:**

**Lab no: 13** **Date:2081/12/28**

**Title: WAP to calculate minimum spanning tree using Prim’s algorithm in C**.

**Prim's Algorithm**

Prim's Algorithm is a greedy algorithm used to find a minimum spanning tree (MST) of a weighted, undirected graph. A minimum spanning tree of a graph is a subset of the edges that connects all the vertices together, without any cycles, and with the minimum possible total edge weight.

**Pseudocode:**

void primMST(int graph[99][99], int V) {

int parent[V];

int key[V];

int mstSet[V];

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

key[i] = INT\_MAX;

mstSet[i] = 0;

}

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet, V);

mstSet[u] = 1;

for (int v = 0; v < V; v++) {

if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

**IDE: VS Code**

**Language: C**

**Source code:**

#include <stdio.h>

#include <limits.h>

int minKey(int key[], int mstSet[], int V); // function prototype

// function to find the vertex with the minimum key value

int minKey(int key[], int mstSet[], int V) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (mstSet[v] == 0 && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

// function to implement prim algorithm

void primMST(int graph[99][99], int V) {

int parent[V]; // array to store the MST

int key[V]; // key values used to pick the minimum weight edge

int mstSet[V]; // to represent the set of vertices included in the MST

// initialize all key values to infinity and mstSet as 0

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

key[i] = INT\_MAX;

mstSet[i] = 0; // All vertices are initially not in MST

}

key[0] = 0; // Start with the first vertex

parent[0] = -1; // The first node is always the root of MST

// Build the MST

for (int count = 0; count < V - 1; count++) {

// Pick the minimum key vertex that is not yet included in MST

int u = minKey(key, mstSet, V);

mstSet[u] = 1; // Add the picked vertex to the MST set

// Update the key values and parent index of the adjacent vertices

for (int v = 0; v < V; v++) {

// graph[u][v] != 0 means there is an edge between u and v

if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

// Print the edges of the MST

printf("Edge \tWeight\n");

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

printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);

}

}

int main() {

int V, E, u, v, w;

// Ask user for the number of vertices and edges

printf("\*\*\tCompiled By Aswin phuyal\t\*\*\n");

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

scanf("%d", &V);

// Create the adjacency matrix for the graph

int graph[99][99] = {0}; // Maximum of 99 vertices, initialize all to 0

// Ask user for the number of edges

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

scanf("%d", &E);

// Input the edges and their weights

printf("Enter each edge in the format (vertex1 vertex2 weight):\n");

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

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

scanf("%d %d %d", &u, &v, &w);

graph[u][v] = graph[v][u] = w; // since graph is undirected

}

// call the prim's algorithm func

primMST(graph, V);

return 0;

}

**Output:**

****

**Lab no: 14** **Date:2081/12/28**

**Title: WAP to search the user input key in the list using linear search in C**.

**Linear search**

Linear search is a simple algorithm used to search for a specific element (key) in a list or array. It works by sequentially checking each element of the list to see if it matches the desired key. The search continues until a match is found or the entire list has been checked.

**Pseudocode**

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

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

if (arr[i] == key) {

return i;

}

}

return -1;

}

**IDE: VS Code**

**Language: C**

**SourceCode:**

#include <stdio.h>

// func to perform linear search

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

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

if (arr[i] == key) {

return i; // return the index if the key is found

}

}

return -1; // return -1 if key is not found

}

int main() {

int n, key;

printf("\*\*\tCompiled By Aswin phuyal\t\*\*\n");

// ask user for the size of the array

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

scanf("%d", &n);

int arr[n]; // declare the array of size n

// ssk user for the elements of the array

printf("Enter the elements of the list: \n");

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

scanf("%d", &arr[i]);

}

// ask user for the key to search

printf("Enter the key to search: ");

scanf("%d", &key);

// perform linear search

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

// display result

if (result == -1) {

printf("Key not found in the list.\n");

} else {

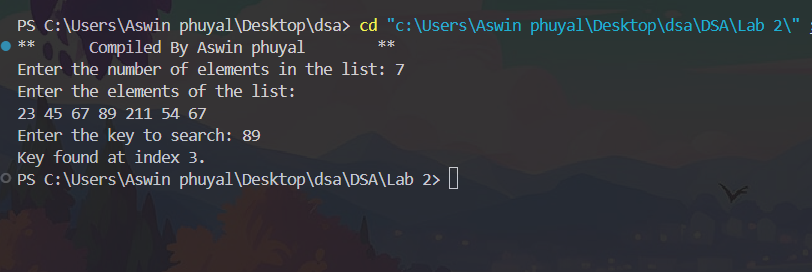
printf("Key found at index %d.\n", result);

}

return 0;

}

**Output:**

****

**Lab no: 15** **Date:2081/12/28**

**Title: WAP to search the user input key in the list using binary search in C**.

**Binary Search**

Binary Search is an efficient algorithm for finding an item from a sorted list or array. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, the search continues in the left half, or if the value is greater, the search continues in the right half. This process repeats until the value is found or the interval is empty.

**Pseudocode:**

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

int left = 0;

int right = n - 1;

while (left <= right) {

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

if (arr[mid] == key) {

return mid;

}

if (arr[mid] < key) {

left = mid + 1;

}

else {

right = mid - 1;

}

}

return -1;

}

**IDE: VS Code**

**Language: C**

**Source code:**

#include <stdio.h>

// func to perform binary search

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

int left = 0;

int right = n - 1;

while (left <= right) {

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

// if key is present at mid

if (arr[mid] == key) {

return mid;

}

// if key is greater than mid ignore left half

if (arr[mid] < key) {

left = mid + 1;

}

// if key is smaller than mid ignore right half

else {

right = mid - 1;

}

}

return -1;

}

int main() {

int n, key;

printf("\*\*\tCompiled By Aswin phuyal\t\*\*\n");

// ask user for no. of elements in array

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

scanf("%d", &n);

int arr[n]; // declare array of size n

// ask user for elements of the array

printf("Enter the elements of the list in sorted order:\n");

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

scanf("%d", &arr[i]);

}

// ask user for the key to search

printf("Enter the key to search: ");

scanf("%d", &key);

// perform binary search

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

// display the result

if (result == -1) {

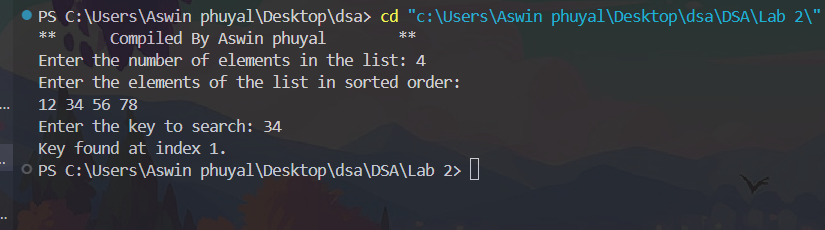
printf("Key not found in the list.\n");

} else {

printf("Key found at index %d.\n", result);

}}

**Output:**

****

**Lab no: 16** **Date:2081/12/28**

**Title: WAP to implement double and quadratic hashing in C**.

**Double Hashing**

Double Hashing is a open addressing collision resolution technique that uses two hash functions to calculate the index where a key should be inserted into the hash table.

new index=(h1(x)+i×h2(x))mod 10

**Quadratic Hashing**

Quadratic Hashing is another open addressing collision resolution technique, but instead of using a secondary hash function to determine the step size, it uses a quadratic function to probe for the next available slot.

new index=(h1(x)+i2)mod 10

**IDE: VS Code**

**Language: C**

**SourceCode:**

#include <stdio.h>

#define TABLE\_SIZE 10

// Hash table arrays

int hashTableDouble[TABLE\_SIZE];

int hashTableQuadratic[TABLE\_SIZE];

// primary hash function

int hash1(int key) {

return key % TABLE\_SIZE;

}

// secondary hash function for Double Hashing

int hash2(int key) {

return 7 - (key % 7);

}

// func to insert into hash table using Double Hashing

void insert\_double\_hashing(int key) {

int index = hash1(key);

int stepSize = hash2(key);

int i = 0;

// Find an empty slot using double hashing

while (hashTableDouble[(index + i \* stepSize) % TABLE\_SIZE] != -1) {

i++;

}

// insert the key into the table at the calculated position

hashTableDouble[(index + i \* stepSize) % TABLE\_SIZE] = key;

}

// function to insert into hash table using Quadratic Hashing

void insert\_quadratic\_hashing(int key) {

int index = hash1(key);

int i = 0;

// find an empty slot using quadratic hashing

while (hashTableQuadratic[(index + i \* i) % TABLE\_SIZE] != -1) {

i++;

}

// insert the key into the table at the calculated position

hashTableQuadratic[(index + i \* i) % TABLE\_SIZE] = key;

}

// function to display hash table contents

void display(int hashTable[], const char \*methodName) {

printf("\nHash Table using %s:\n", methodName);

for (int i = 0; i < TABLE\_SIZE; i++) {

if (hashTable[i] != -1) {

printf("%d -> %d\n", i, hashTable[i]);

} else {

printf("%d -> Empty\n", i);

}

}

}

int main() {

printf("\*\*\tCompiled By Aswin phuyal\t\*\*\n");

int key;

// initialize hash tables with -1 to indicate empty slots

for (int i = 0; i < TABLE\_SIZE; i++) {

hashTableDouble[i] = -1;

hashTableQuadratic[i] = -1;

}

// take input

printf("Enter keys to insert into the hash table (enter -1 to stop):\n");

while (1) {

printf("Enter key: ");

scanf("%d", &key);

if (key == -1) {

break;

}

// insert the key into both hash tables

insert\_double\_hashing(key);

insert\_quadratic\_hashing(key);

}

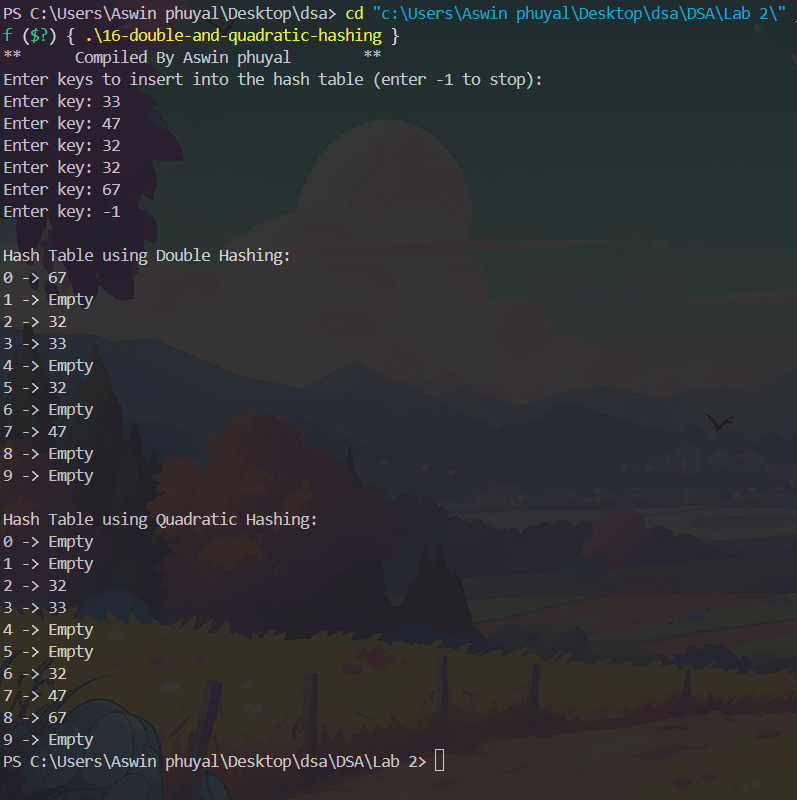
// display

display(hashTableDouble, "Double Hashing");

display(hashTableQuadratic, "Quadratic Hashing");

}

**Output:**

****

**Lab no:18**  **Date:2081/12/28**

**Title: WAP to sort data using selection sort in C**.

**Selection Sort**

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and swapping it with the first unsorted element. The algorithm divides the list into two parts: sorted and unsorted. Initially, the sorted part is empty, and the unsorted part is the entire list. With each iteration, the algorithm adds one element to the sorted part.

**Pseudocode:**

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

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

int minIndex = i;

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

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

if (minIndex != i) {

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

**IDE: VS Code**

**Language: C**

**Sourcecode:**

#include <stdio.h>

// Function to perform selection sort

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

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

int minIndex = i;

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

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap smallest element with the first unsorted element

if (minIndex != i) {

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

// Print array after each pass

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

for (int k = 0; k < n; k++) {

printf("%d ", arr[k]);

}

printf("\n");

}

}

// Function to print the array

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

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

printf("\*\*\tCompiled By Aswin Phuyal\t\*\*\n");

int n;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

}

printf("\nOriginal array: ");

printArray(arr, n);

selectionSort(arr, n);

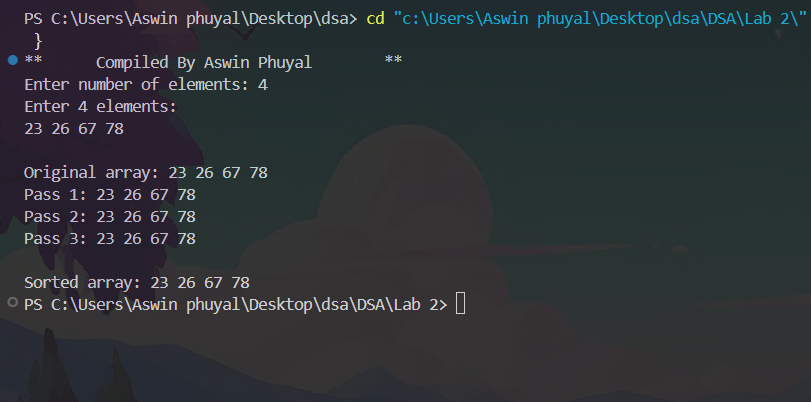
printf("\nSorted array: ");

printArray(arr, n);

return 0;

}

**Output:**

****