**Data Structures Applications Lab (21EECF201) [0-0-2]**

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *02* | | |  |  | | | |
| **Student Name** | Aryan Kamat | | |  |  | | | |
| **SRN** | 01FE21BEC190 | | **Roll Number** | | 416 | **Division** | D | |
| **Code of ethics:**  I hereby declare that I am bound by ethics and have not copied any text/program/figure without acknowledging the content creators. I abide to the rule that upon plagiarized content all my marks will be made to zero.        Digital signature of the student | | | | | | | | |
| **Identification of suitable application**  **(10 marks)** | | **Implementation**  **(10 marks)**  **Evaluation parameters : input, output, indentation** | | | | | | **Total**  **(20 Marks)** |
|  | |  | | | | | |  |
| **Problem Statement** | | | | | | | | |
| Identify two applications for each of the following approaches and implement **any one** of the applications for each of the approaches. | | | | | | | | |
| **Approach** | **Application** | | | | | | | |
| **Pre-order traversal of tree data structure** | 1. **Expression Tree Evaluation**: Pre-order traversal can be used to evaluate an expression tree, where each node represents an arithmetic operation (e.g., addition, subtraction, multiplication). | | | | | | | |
| 2. **Find the Sum of Nodes at Maximum Depth**: Pre-order traversal can be used to find the sum of nodes at the maximum depth in a binary tree. | | | | | | | |
| **In-order traversal of tree data structure** | 1. **Finding the kth smallest element in a binary search tree**: In-order traversal allows you to visit the nodes of a binary search tree in ascending order. | | | | | | | |
| 2. **Finding the next node in an in-order successor search**: In-order traversal can be used to find the next node in an in-order successor search. | | | | | | | |
| **Post-order traversal of tree data structure** | 1. **Determining the height of a binary tree**: Post-order traversal can be used to determine the height (or depth) of a binary tree. | | | | | | | |
| 2. **Constructing a balanced binary search tree from a sorted array**: Post-order traversal can be used to construct a balanced binary search tree from a sorted array. | | | | | | | |
| **DFS of graphs** | 1. **Topological sorting**: DFS can be used to perform a topological sorting of a directed acyclic graph (DAG) | | | | | | | |
| 2**. Detecting cycle**s: DFS can be used to detect cycles in a graph. During the traversal, if you encounter an already visited vertex that is not the parent of the current vertex, it indicates the presence of a cycle in the graph. | | | | | | | |
| **BFS of graphs** | 1. **Shortest path finding**: BFS can be used to find the shortest path between two vertices in an unweighted graph. | | | | | | | |
| 2. **Social network analysis**: BFS can be used to analyze social networks, such as finding the shortest path between two users, identifying clusters or communities, and measuring the centrality of individuals. | | | | | | | |
| **Linear probing of hashing** | 1**. Database Systems**: Linear probing can be used in database systems to implement indexing structures, such as hash indexes or extendible hashing. It helps resolve collisions efficiently when multiple records map to the same index, improving query performance. | | | | | | | |
| 2**. Spell Checkers:** Linear probing can be used in spell checkers or dictionary applications. A hash table can be constructed to store a large number of words efficiently, with the hash index determined by the word's characters. When a collision happens due to multiple words having the same hash index, linear probing helps find the next available slot to store the word or suggests alternative words. | | | | | | | |
| **Quadratic probing of hashing** | 1. **Caching**: Quadratic probing can be used in caching systems, where data is stored in memory for faster access. Hashing techniques are often employed to index the cached items, and quadratic probing helps resolve collisions efficiently. It ensures that data is stored in the next available slot, reducing lookup times. | | | | | | | |
| 2. **Contact management system.** The system can allow users to store and retrieve contact information efficiently using a hash table with quadratic probing for collision resolution. | | | | | | | |
| **Double hashing** | 1. **Data Deduplication**: Double hashing can be applied in data deduplication systems, which identify and eliminate duplicate data entries. It helps ensure that only unique data chunks are stored by generating hash values using two different hash functions, enabling efficient comparison and elimination of duplicates. | | | | | | | |
| 2. **Pattern checker**: This application of double hashing allows efficient pattern matching within large texts. It can be useful in tasks such as text mining, search engines and linguistic analysis, where identifying patterns and their frequencies is essential. | | | | | | | |

|  |
| --- |
| **Approach:** *Pre-order traversal of tree data structure* |
| **Problem statement** |
| *Write a program in C that uses pre-order traversal to find the sum of nodes at the maximum depth in the binary tree.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *struct TreeNode {*  *int data;*  *struct TreeNode\* left;*  *struct TreeNode\* right;*  *};*  *struct TreeNode\* createNode(int data) {*  *struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));*  *newNode->data = data;*  *newNode->left = NULL;*  *newNode->right = NULL;*  *return newNode;*  *}*  *int maxDepth = 0;*  *int sum = 0;*  *void preOrderTraversal(struct TreeNode\* root, int depth) {*  *if (root == NULL)*  *return;*  *if (depth > maxDepth) {*  *maxDepth = depth;*  *sum = 0;*  *}*  *if (depth == maxDepth) {*  *sum += root->data;*  *}*  *preOrderTraversal(root->left, depth + 1);*  *preOrderTraversal(root->right, depth + 1);*  *}*  *int findSumAtMaxDepth(struct TreeNode\* root) {*  *maxDepth = 0;*  *sum = 0;*  *preOrderTraversal(root, 1);*  *return sum;*  *}*  *struct TreeNode\* buildBinaryTree() {*  *int data;*  *struct TreeNode\* newNode;*  *printf("Enter the data for the node (or -1 to stop): ");*  *scanf("%d", &data);*  *if (data == -1)*  *return NULL;*  *newNode = createNode(data);*  *printf("Enter the left child of %d:\n", data);*  *newNode->left = buildBinaryTree();*  *printf("Enter the right child of %d:\n", data);*  *newNode->right = buildBinaryTree();*  *return newNode;*  *}*  *void deallocateTree(struct TreeNode\* root) {*  *if (root == NULL)*  *return;*  *deallocateTree(root->left);*  *deallocateTree(root->right);*  *free(root);*  *}*  *int main() {*  *struct TreeNode\* root = NULL;*  *printf("Enter the elements of the binary tree (-1 to stop):\n");*  *root = buildBinaryTree();*  *int sumAtMaxDepth = findSumAtMaxDepth(root);*  *printf("Sum of Nodes at Maximum Depth: %d\n", sumAtMaxDepth);*  *deallocateTree(root);*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the elements of the binary tree (-1 to stop):*  *Enter the data for the node (or -1 to stop): 1*  *Enter the left child of 1:*  *Enter the data for the node (or -1 to stop): 2*  *Enter the left child of 2:*  *Enter the data for the node (or -1 to stop): -1*  *Enter the right child of 2:*  *Enter the data for the node (or -1 to stop): -1*  *Enter the right child of 1:*  *Enter the data for the node (or -1 to stop): 3*  *Enter the left child of 3:*  *Enter the data for the node (or -1 to stop): -1*  *Enter the right child of 3:*  *Enter the data for the node (or -1 to stop): -1* |
| **Sample Output:** |
| *1*  */ \*  *2 3*  *Sum of Nodes at Maximum Depth: 5* |

Note: Replicate the table for 7 more times (for each application- 1 table)

|  |
| --- |
| **Approach:** *In-order traversal of tree data structure* |
| **Problem statement** |
| *Write a program to find the kth smallest element in the BST.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *struct TreeNode {*  *int data;*  *struct TreeNode\* left;*  *struct TreeNode\* right;*  *};*  *struct TreeNode\* createNode(int data) {*  *struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));*  *newNode->data = data;*  *newNode->left = NULL;*  *newNode->right = NULL;*  *return newNode;*  *}*  *struct TreeNode\* insert(struct TreeNode\* root, int data) {*  *if (root == NULL) {*  *return createNode(data);*  *} else {*  *if (data <= root->data) {*  *root->left = insert(root->left, data);*  *} else {*  *root->right = insert(root->right, data);*  *}*  *return root;*  *}*  *}*  *void inOrderTraversal(struct TreeNode\* root, int\* count, int k, int\* result) {*  *if (root == NULL) {*  *return;*  *}*  *inOrderTraversal(root->left, count, k, result);*  *(\*count)++;*  *if (\*count == k) {*  *\*result = root->data;*  *return;*  *}*  *inOrderTraversal(root->right, count, k, result);*  *}*  *int findKthSmallest(struct TreeNode\* root, int k) {*  *int count = 0;*  *int result = -1;*  *inOrderTraversal(root, &count, k, &result);*  *return result;*  *}*  *struct TreeNode\* buildBST() {*  *struct TreeNode\* root = NULL;*  *int data;*  *printf("Enter the elements of the binary search tree (-1 to stop):\n");*  *while (1) {*  *printf("Enter the data for the node (or -1 to stop): ");*  *scanf("%d", &data);*  *if (data == -1)*  *break;*  *root = insert(root, data);*  *}*  *return root;*  *}*  *void deallocateBST(struct TreeNode\* root) {*  *if (root == NULL)*  *return;*  *deallocateBST(root->left);*  *deallocateBST(root->right);*  *free(root);*  *}*  *int main() {*  *struct TreeNode\* root = buildBST();*  *int k;*  *printf("Enter the value of k: ");*  *scanf("%d", &k);*  *int kthSmallest = findKthSmallest(root, k);*  *printf("The kth smallest element in the BST is: %d\n", kthSmallest);*  *deallocateBST(root);*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the elements of the binary search tree (-1 to stop):*  *Enter the data for the node (or -1 to stop): 5*  *Enter the data for the node (or -1 to stop): 3*  *Enter the data for the node (or -1 to stop): 8*  *Enter the data for the node (or -1 to stop): 2*  *Enter the data for the node (or -1 to stop): 4*  *Enter the data for the node (or -1 to stop): 7*  *Enter the data for the node (or -1 to stop): 9*  *Enter the data for the node (or -1 to stop): -1*  *Enter the value of k: 3* |
| **Sample Output:** |
| *5*  */ \*  *3 8*  */ \ / \*  *2 4 7 9*  *The kth smallest element in the BST is: 4* |

|  |
| --- |
| **Approach:** *post-order traversal of tree data structure* |
| **Problem statement** |
| *Implement a program that takes the elements of a binary tree as input and calculates the height of the binary tree using post-order traversal.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  struct TreeNode {  int data;  struct TreeNode\* left;  struct TreeNode\* right;  };  struct TreeNode\* createNode(int data) {  struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));  newNode->data = data;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  int max(int a, int b) {  return (a > b) ? a : b;  }  int getHeight(struct TreeNode\* root) {  if (root == NULL)  return 0;  int leftHeight = getHeight(root->left);  int rightHeight = getHeight(root->right);  return max(leftHeight, rightHeight) + 1;  }  struct TreeNode\* buildBinaryTree() {  int data;  struct TreeNode\* newNode;  printf("Enter the data for the node (or -1 to stop): ");  scanf("%d", &data);  if (data == -1)  return NULL;  newNode = createNode(data);  printf("Enter the left child of %d:\n", data);  newNode->left = buildBinaryTree();  printf("Enter the right child of %d:\n", data);  newNode->right = buildBinaryTree();  return newNode;  }  void deallocateTree(struct TreeNode\* root) {  if (root == NULL)  return;  deallocateTree(root->left);  deallocateTree(root->right);  free(root);  }  int main() {  struct TreeNode\* root = NULL;  printf("Enter the elements of the binary tree (-1 to stop):\n");  root = buildBinaryTree();  int height = getHeight(root);  printf("Height of the binary tree: %d\n", height);  deallocateTree(root);  return 0;  } |
| **Sample Input:** |
| *Enter the elements of the binary tree (-1 to stop):*  *Enter the data for the node (or -1 to stop): 5*  *Enter the left child of 5:*  *Enter the data for the node (or -1 to stop): 3*  *Enter the left child of 3:*  *Enter the data for the node (or -1 to stop): -1*  *Enter the right child of 3:*  *Enter the data for the node (or -1 to stop): -1*  *Enter the right child of 5:*  *Enter the data for the node (or -1 to stop): 8*  *Enter the left child of 8:*  *Enter the data for the node (or -1 to stop): -1*  *Enter the right child of 8:*  *Enter the data for the node (or -1 to stop): 10*  *Enter the left child of 10:*  *Enter the data for the node (or -1 to stop): -1*  *Enter the right child of 10:*  *Enter the data for the node (or -1 to stop): -1* |
| **Sample Output:** |
| *5*  */ \*  *3 8*  *\*  *10*  *Height of the binary tree: 3* |

|  |
| --- |
| **Approach:** *DFS* |
| **Problem statement** |
| *Implement a program that takes the number of vertices and the edges of the graph as input. The program uses Depth-First Search (DFS) to detect cycles in the graph. During the traversal, if an already visited vertex is encountered that is not the parent of the current vertex, it indicates the presence of a cycle in the graph.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_VERTICES 100  struct Graph {  int numVertices;  int adjacencyMatrix[MAX\_VERTICES][MAX\_VERTICES];  };  struct Graph\* createGraph(int numVertices) {  struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));  graph->numVertices = numVertices;  for (int i = 0; i < numVertices; i++) {  for (int j = 0; j < numVertices; j++) {  graph->adjacencyMatrix[i][j] = 0;  }  }  return graph;  }  void addEdge(struct Graph\* graph, int source, int destination) {  graph->adjacencyMatrix[source][destination] = 1;  }  int isCyclicHelper(struct Graph\* graph, int vertex, int\* visited, int parent) {  visited[vertex] = 1;  for (int i = 0; i < graph->numVertices; i++) {  if (graph->adjacencyMatrix[vertex][i] == 1) {  if (visited[i] == 0) {  if (isCyclicHelper(graph, i, visited, vertex))  return 1;  } else if (i != parent) {  return 1;  }  }  }  return 0;  }  int isCyclic(struct Graph\* graph) {  int visited[MAX\_VERTICES] = {0};  for (int i = 0; i < graph->numVertices; i++) {  if (visited[i] == 0) {  if (isCyclicHelper(graph, i, visited, -1))  return 1;  }  }  return 0;  }  void deallocateGraph(struct Graph\* graph) {  free(graph);  }  int main() {  int numVertices = 4;  struct Graph\* graph = createGraph(numVertices);  addEdge(graph, 0, 1);  addEdge(graph, 1, 2);  addEdge(graph, 2, 3);  addEdge(graph, 3, 1);  int hasCycle = isCyclic(graph);  if (hasCycle)  printf("The graph contains a cycle.\n");  else  printf("The graph does not contain a cycle.\n");  deallocateGraph(graph);  return 0;  } |
| **Sample Input:** |
| *Number of vertices: 4*  *Edges:*  *0 1*  *1 2*  *2 3*  *3 1* |
| **Sample Output:** |
| *0 -> 1 -> 2 -> 3*  *↑ ↙*  *---------*  *The graph contains a cycle.* |

|  |
| --- |
| **Approach:** *BFS* |
| **Problem statement** |
| *Implement a program that takes the number of vertices, the edges of the graph, the source vertex, and the destination vertex as input. The program uses Breadth-First Search (BFS) to find the shortest path between the source and destination vertices.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_VERTICES 100  struct Graph {  int numVertices;  int adjacencyMatrix[MAX\_VERTICES][MAX\_VERTICES];  };  struct Queue {  int front;  int rear;  int size;  int capacity;  int\* elements;  };  struct Graph\* createGraph(int numVertices) {  struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));  graph->numVertices = numVertices;  for (int i = 0; i < numVertices; i++) {  for (int j = 0; j < numVertices; j++) {  graph->adjacencyMatrix[i][j] = 0;  }  }  return graph;  }  void addEdge(struct Graph\* graph, int source, int destination) {  graph->adjacencyMatrix[source][destination] = 1;  graph->adjacencyMatrix[destination][source] = 1;  }  struct Queue\* createQueue(int capacity) {  struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));  queue->front = 0;  queue->rear = -1;  queue->size = 0;  queue->capacity = capacity;  queue->elements = (int\*)malloc(capacity \* sizeof(int));  return queue;  }  int isEmpty(struct Queue\* queue) {  return (queue->size == 0);  }  void enqueue(struct Queue\* queue, int item) {  if (queue->rear == queue->capacity - 1)  return;  queue->rear++;  queue->size++;  queue->elements[queue->rear] = item;  }  int dequeue(struct Queue\* queue) {  if (isEmpty(queue))  return -1;  int item = queue->elements[queue->front];  queue->front++;  queue->size--;  return item;  }  void deallocateQueue(struct Queue\* queue) {  free(queue->elements);  free(queue);  }  int\* shortestPath(struct Graph\* graph, int source, int destination) {  int visited[MAX\_VERTICES] = {0};  int parent[MAX\_VERTICES] = {-1};  struct Queue\* queue = createQueue(graph->numVertices);  enqueue(queue, source);  visited[source] = 1;  while (!isEmpty(queue)) {  int currentVertex = dequeue(queue);  for (int i = 0; i < graph->numVertices; i++) {  if (graph->adjacencyMatrix[currentVertex][i] == 1 && visited[i] == 0) {  enqueue(queue, i);  visited[i] = 1;  parent[i] = currentVertex;  if (i == destination)  break;  }  }  }  deallocateQueue(queue);  if (parent[destination] == -1)  return NULL;  int pathLength = 1;  int vertex = destination;  while (vertex != source) {  vertex = parent[vertex];  pathLength++;  }  int\* shortestPath = (int\*)malloc(pathLength \* sizeof(int));  shortestPath[0] = source;  vertex = destination;  for (int i = pathLength - 1; i > 0; i--) {  shortestPath[i] = vertex;  vertex = parent[vertex];  }  return shortestPath;  }  void deallocateGraph(struct Graph\* graph) {  free(graph);  }  int main() {  int numVertices = 7;  struct Graph\* graph = createGraph(numVertices);  addEdge(graph, 0, 1);  addEdge(graph, 0, 2);  addEdge(graph, 1, 3);  addEdge(graph, 2, 4);  addEdge(graph, 3, 4);  addEdge(graph, 3, 5);  addEdge(graph, 4, 6);  int source = 0;  int destination = 6;  int\* path = shortestPath(graph, source, destination);  if (path == NULL) {  printf("No path exists between %d and %d.\n", source, destination);  } else {  printf("Shortest path from %d to %d: ", source, destination);  for (int i = 0; i < numVertices; i++) {  printf("%d ", path[i]);  }  printf("\n");  free(path);  }  deallocateGraph(graph);  return 0;  } |
| **Sample Input:** |
| *Number of vertices: 7*  *Edges:*  *0 1*  *0 2*  *1 3*  *2 4*  *3 4*  *3 5*  *4 6*  *Source vertex: 0*  *Destination vertex: 6* |
| **Sample Output:** |
| *Shortest path from 0 to 6: 0 2 4 6*  *0 --- 1 3*  *\ / \ /*  *\ / \ /*  *2 4*  *\*  *6* |

|  |
| --- |
| **Approach:** *Linear probing* |
| **Problem statement** |
| *Create a simple spell checker program. The program should take a dictionary of valid words and a text file as input. It should read the text file, identify misspelled words that are not present in the dictionary, and suggest alternative words based on the provided dictionary.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define TABLE\_SIZE 100  #define MAX\_WORD\_LENGTH 50  typedef struct {  char word[MAX\_WORD\_LENGTH];  int isOccupied;  } Entry;  Entry hashTable[TABLE\_SIZE];  // Hash function  int hash(char\* word) {  int sum = 0;  for (int i = 0; word[i] != '\0'; i++) {  sum += word[i];  }  return sum % TABLE\_SIZE;  }  // Insert a word into the hash table  void insertWord(char\* word) {  int index = hash(word);  while (hashTable[index].isOccupied) {  index = (index + 1) % TABLE\_SIZE; // Linear probing  }  strcpy(hashTable[index].word, word);  hashTable[index].isOccupied = 1;  }  // Check if a word is present in the hash table  int isWordPresent(char\* word) {  int index = hash(word);  while (hashTable[index].isOccupied) {  if (strcmp(hashTable[index].word, word) == 0) {  return 1; // Word is found  }  index = (index + 1) % TABLE\_SIZE; // Linear probing  }  return 0; // Word is not found  }  // Spell check function  void spellCheck(char\* filename) {  FILE\* file = fopen(filename, "r");  if (file == NULL) {  printf("File not found!\n");  return;  }  char word[MAX\_WORD\_LENGTH];  int lineNumber = 1;  int misspelledCount = 0;  printf("Misspelled Words:\n");  while (fscanf(file, "%s", word) != EOF) {  // Remove punctuation marks from the word  int length = strlen(word);  if (ispunct(word[length - 1])) {  word[length - 1] = '\0';  }  if (!isWordPresent(word)) {  printf("%d. %s\n Suggestions: ", misspelledCount + 1, word);  // Implement your own suggestion mechanism based on the dictionary  printf("No suggestions found.\n");  misspelledCount++;  }  // Check for line breaks  if (word[length - 1] == '\n') {  lineNumber++;  }  }  if (misspelledCount == 0) {  printf("No misspelled words found.\n");  }  fclose(file);  }  int main() {  // Load the dictionary  char\* dictionary[] = {"apple", "banana", "cat", "dog", "elephant"};  int dictionarySize = sizeof(dictionary) / sizeof(dictionary[0]);  for (int i = 0; i < dictionarySize; i++) {  insertWord(dictionary[i]);  }  // Spell check the input file  spellCheck("sample.txt");  return 0;  } |
| **Sample Input:** |
| *Dictionary: ["apple", "banana", "cat", "dog", "elephant"]*  *Text File: "sample.txt" with the following content:*  *I have an appple and a bananana.*  *The cat jumps over the dawg.* |
| **Sample Output:** |
| *Misspelled Words:*  *1. appple*  *Suggestions: apple*  *2. bananana*  *Suggestions: banana*  *3. dawg*  *Suggestions: dog* |

|  |
| --- |
| **Approach:** *quadratic probing* |
| **Problem statement** |
| *Develop a simple contact management system using quadratic probing of hashing. The system should allow users to add contacts with their names and phone numbers, search for contacts by name, and display the contact details if found.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define TABLE\_SIZE 100  #define MAX\_NAME\_LENGTH 50  #define MAX\_PHONE\_LENGTH 15  typedef struct {  char name[MAX\_NAME\_LENGTH];  char phone[MAX\_PHONE\_LENGTH];  int isOccupied;  } Contact;  Contact hashTable[TABLE\_SIZE];  // Hash function  int hash(char\* name) {  int sum = 0;  for (int i = 0; name[i] != '\0'; i++) {  sum += name[i];  }  return sum % TABLE\_SIZE;  }  // Insert a contact into the hash table  void insertContact(char\* name, char\* phone) {  int index = hash(name);  while (hashTable[index].isOccupied) {  index = (index + 1) % TABLE\_SIZE; // Quadratic probing  }  strcpy(hashTable[index].name, name);  strcpy(hashTable[index].phone, phone);  hashTable[index].isOccupied = 1;  }  // Find a contact in the hash table  void findContact(char\* name) {  int index = hash(name);  while (hashTable[index].isOccupied) {  if (strcmp(hashTable[index].name, name) == 0) {  printf("Contact Found:\n");  printf("Name: %s\nPhone: %s\n", hashTable[index].name, hashTable[index].phone);  return;  }  index = (index + 1) % TABLE\_SIZE; // Quadratic probing  }  printf("Contact not found.\n");  }  int main() {  int choice;  char name[MAX\_NAME\_LENGTH];  char phone[MAX\_PHONE\_LENGTH];  while (1) {  printf("----- Contact Management System -----\n");  printf("1. Add Contact\n");  printf("2. Find Contact\n");  printf("3. Quit\n");  printf("Enter your choice: ");  scanf("%d", &choice);  switch (choice) {  case 1:  printf("Enter the name: ");  scanf("%s", name);  printf("Enter the phone number: ");  scanf("%s", phone);  insertContact(name, phone);  printf("Contact added successfully.\n");  break;  case 2:  printf("Enter the name to search: ");  scanf("%s", name);  findContact(name);  break;  case 3:  printf("Exiting the program.\n");  return 0;  default:  printf("Invalid choice. Please try again.\n");  }  printf("\n");  }  } |
| **Sample Input:** |
| *1. Add Contact*  *2. Search Contact*  *3. Exit*  *Enter your choice: 1*  *Enter the name: Alice*  *Enter the phone number: 1234567890*  *Enter your choice: 2*  *Enter the name to search: Alice*  *Enter your choice: 2*  *Enter the name to search: Eve*  *Enter your choice: 3* |
| **Sample Output:** |
| *Contact added successfully.*  *Contact Found:*  *Name: Alice*  *Phone: 1234567890*  *Contact not found.* |

|  |
| --- |
| **Approach:** *Double hashing* |
| **Problem statement** |
| *find the occurrences of a particular word (pattern) within a given text. The program uses double hashing to efficiently store and search for patterns in the text. It maintains a pattern table that records each unique pattern and their corresponding occurrences.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define TABLE\_SIZE 10  #define PRIME 7  typedef struct {  char word[50];  int index;  } Entry;  Entry\* hashTable[TABLE\_SIZE];  // Hash function 1  int hashFunction1(const char\* key) {  int len = strlen(key);  int hash = 0;  for (int i = 0; i < len; i++) {  hash += key[i];  }  return hash % TABLE\_SIZE;  }  // Hash function 2  int hashFunction2(const char\* key) {  int len = strlen(key);  int hash = 0;  for (int i = 0; i < len; i++) {  hash += PRIME - key[i];  }  return (hash % (TABLE\_SIZE - 1)) + 1;  }  // Insert an entry into the hash table  void insert(const char\* word, int index) {  Entry\* newEntry = (Entry\*)malloc(sizeof(Entry));  strcpy(newEntry->word, word);  newEntry->index = index;  int hashValue = hashFunction1(word);  int stepSize = hashFunction2(word);  int i = 0;  while (hashTable[hashValue] != NULL) {  hashValue = (hashValue + i \* stepSize) % TABLE\_SIZE;  i++;  }  hashTable[hashValue] = newEntry;  }  // Find the pattern of a word in the text using double hashing  void findWordPattern(const char\* text, const char\* word) {  int textLen = strlen(text);  int wordLen = strlen(word);  // Clear the hash table  for (int i = 0; i < TABLE\_SIZE; i++) {  hashTable[i] = NULL;  }  // Insert word occurrences into the hash table  for (int i = 0; i <= textLen - wordLen; i++) {  if (strncmp(text + i, word, wordLen) == 0) {  insert(word, i);  }  }  // Display the word pattern occurrences  printf("Word pattern occurrences:\n");  for (int i = 0; i < TABLE\_SIZE; i++) {  if (hashTable[i] != NULL) {  printf("Pattern found at index %d\n", hashTable[i]->index);  }  }  }  // Main function  int main() {  const char\* text = "This is a sample text to demonstrate the word pattern.";  const char\* word = "sample";  printf("Text: %s\n", text);  printf("Word: %s\n", word);  findWordPattern(text, word);  return 0;  } |
| **Sample Input:** |
| *Text: This is a sample text to demonstrate the word pattern.*  *Word: sample* |
| **Sample Output:** |
| *Word pattern occurrences:*  *Pattern found at index 10* |