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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *02* | | |  |  | | | |
| **Student Name** | SAFA IMTIHAZ SAYYAD | | |  |  | | | |
| **SRN** | 01FE21BEC205 | | **Roll Number** | | 429 | **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.To Build Decision Trees:  Pre-order traversal can be used to traverse decision trees. These are commonly used in machine learning for classification and regression tasks. By traversing the decision tree in pre-order, you can follow the sequence of decisions and conditions to reach a specific outcome or prediction. | | | | | | | |
| 2.Duplicating a Tree:  Pre-order traversal can be used to duplicate a tree. By traversing the original tree in pre-order and creating a new node for each visited node, identical copy of the original tree can be generated. | | | | | | | |
| **In-order traversal of tree data structure** | 1. Finding Median in a Binary Search Tree:  In-order traversal can be used to find the median value in a binary search tree. By performing an in-order traversal and counting the number of nodes visited, you can determine the median element in the tree without traversing the entire tree as in-order tree **gives increasing order** of the elements of the nodes. | | | | | | | |
| 2. Finding the kth Element in a BST:  In-order traversal can be used to find the kth smallest or kth largest element in a BST in an efficient manner. By traversing the tree in in-order and maintaining a count of visited nodes, you can identify the kth element without having to traverse the entire tree. | | | | | | | |
| **Post-order traversal of tree data structure** | 1. Simplifying Expressions:  Post-order traversal can be used to simplify complex expressions or algebraic equations. By traversing the expression tree in post-order and applying simplification rules, the expression can be reduced to its simplest form. | | | | | | | |
| 2. Binary Tree Rebalancing:  Post-order traversal can be used to rebalance binary trees, such as AVL trees or Red-Black trees. By performing post-order traversal, necessary rotations or adjustments can be identified/performed to rebalance the tree and maintain its desired properties. | | | | | | | |
| **DFS of graphs** | 1. Strongly Connected Components (SCCs):  DFS is used to find strongly connected components in a directed graph. SCCs are subgraphs in which there is a directed path between any pair of vertices. By performing DFS on the graph and tracking back edges, you can identify and group vertices into SCCs used in processes like analysing the structure of networks, graph mining, and compiler optimizations. | | | | | | | |
| 2. Detecting Cycles in Undirected Graphs*- Given an undirected graph, determine if it contains any cycles.*:  DFS can be used to detect cycles in an undirected graph. By tracking back edges during the DFS 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. This is useful in cycle detection, deadlock detection, and identifying circular dependencies in systems. | | | | | | | |
| **BFS of graphs** | 1. Navigation Systems and Routing Algorithms- *Helping customers to find shortest path from source to destination so that they find Uber in a cost efficient way*:  BFS can be used in navigation systems and routing algorithms to find the shortest path between two locations in a road network. By representing the road network as an unweighted graph, with vertices as intersections and edges as roads connecting them, BFS can be used to compute the shortest path for finding routes. | | | | | | | |
| 2. Network Broadcast:  BFS can be used to simulate network broadcasts, where a message or information needs to be sent from one node to all other nodes in the network. By starting the BFS traversal from the source node and visiting adjacent nodes level by level, BFS makes sure that the broadcast message reaches all reachable nodes in the most efficient way. This is commonly used in network communication protocols, message dissemination, and etc. | | | | | | | |
| **Linear probing of hashing** | 1. Database Indexing:  Linear probing can be utilized in database indexing, where hash tables are used to store and retrieve data efficiently. In database systems, indexes are created on specific columns of a table to speed up search operations. When a collision occurs during indexing, linear probing allows for finding the next available slot in the index table to store the collided entry. This ensures that the index remains efficient, as the search for a specific value can be quickly narrowed down by following the linear probe sequence until the desired entry is found or an empty slot is encountered. | | | | | | | |
| 2. Efficient Insertion and Retrieval - *Implementing a simple hashing system using separate chaining to efficiently insert and retrieve data elements:*  Linear hashing ensures that inserting and retrieving records remains efficient, even as the database grows. The dynamic expansion of the hash table helps maintain a balanced load factor, reducing the likelihood of collisions and preserving fast access times. | | | | | | | |
| **Quadratic probing of hashing** | 1. Compiler Symbol Tables - *Designing a compiler symbol table system for real-life programming languages. The goal is to develop a data structure and operations that efficiently handle symbol table management during the compilation process:*  In compiler design, symbol tables are used to store information about identifiers (e.g., variables, functions, classes) in a program. Quadratic probing can be applied to resolve collisions when different identifiers hash to the same index in the symbol table. This ensures efficient retrieval and management of symbol table entries. | | | | | | | |
| 2. Cryptographic Algorithms:  Quadratic probing can be utilized in cryptographic algorithms that rely on hash functions for data integrity checks or password storage. It provides an efficient method for handling collisions and maintaining the integrity of the hashed data. | | | | | | | |
| **Double hashing** | 1. Cryptographic Hash Functions:  Cryptographic hash functions are widely used in various security applications, including data integrity verification, password storage, digital signatures, and message authentication. They take an input (message or data) and produce a fixed-size hash value or digest. | | | | | | | |
| 2. Open Addressing Hash Tables - *Design a phone directory system that utilizes double hashing to efficiently store and retrieve phone numbers and associated contact information. The system should provide fast access times, handle collisions, and allow users to easily search and update phone records:*  Double hashing is commonly used as a collision resolution technique in open addressing hash tables. In open addressing, all key-value pairs are stored directly in the hash table array, without the use of additional data structures like linked lists or separate chaining. | | | | | | | |

|  |
| --- |
| **Approach: PRE-ORDER TRAVERSAL OF TREE DATA STRUCTURE** |
| **Problem statement** |
| Problem statement: *Duplicating a Tree*  Reason of approach: Pre-order traversal can be used to duplicate a tree due to the ease of its implementation They are pretty straightforward to implement and the logic is intuitive, as we first process the current node, then recursively traverse the left subtree, and lastly the right subtree. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  typedef struct node // Structure for binary tree node  {  int data;  struct node\* left;  struct node\* right;  } node;  node\* createnode(int data)  {  node\* newnode = (node\*)malloc(sizeof(node)); //create a new node  if (newnode == NULL)  {  printf("Memory allocation error!\n");  exit(1);  }  newnode->data = data;  newnode->left = NULL;  newnode->right = NULL;  return newnode;  }  node\* cloneBinaryTree(Node\* root)  {  if (root == NULL)  {  return NULL;  }  node\* clonenode = createnode(root->data); //creation of duplicate node  clonenode->left = cloneBinaryTree(root->left); //pre order traversal of duplicate node  clonenode->right = cloneBinaryTree(root->right);  return clonenode;  }  void preOrderTraversal(Node\* root)  {  if (root == NULL)  {  return;  }  printf("%d ", root->data); //pre-order print a binary tree  preOrderTraversal(root->left);  preOrderTraversal(root->right);  }  int main()  {  node\* root = createnode(1); // create the binary tree  root->left = createnode(2);  root->right = createnode(3);  root->left->left = createnode(4);  root->left->right = createnode(5);  node\* clonedTree = cloneBinaryTree(root); // clone the binary tree  printf("Original Tree: "); // print the original and cloned trees using pre-order traversal  preOrderTraversal(root);  printf(" Duplicate Tree: ");  preOrderTraversal(clonedTree);  return 0;  } |
| **Sample Input:** |
| Original Tree: 1 2 4 5 3 |
| **Sample Output:** |
| Duplicate Tree: 1 2 4 5 3 |

|  |
| --- |
| **Approach: IN-ORDER TRAVERSAL OF TREE DATA STRUCTURE** |
| **Problem statement** |
| Problem statement: *Finding Median in a Binary Search Tree*  Reason of approach: For determining the median, we need to determine the in-order of the BST because in in-order the elements of the nodes will be sorted, thus determining the median becomes an easier task. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  typedef struct Node // Structure for BST node  {  int data;  struct Node\* left;  struct Node\* right;  } Node;  Node\* createNode(int data) // Function to create a new node  {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (newNode == NULL)  {  printf("Memory allocation error!\n");  exit(1);  }  newNode->data = data;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  Node\* insertNode(Node\* root, int data) // Function to insert a node in BST  {  if (root == NULL)  {  return createNode(data);  }  if (data < root->data)  {  root->left = insertNode(root->left, data);  }  else if (data > root->data)  {  root->right = insertNode(root->right, data);  }  return root;  }  void findMedian(Node\* root, int\* count, int\* median, int targetCount)  {  if (root == NULL) // Function to perform inorder traversal and find the median  {  return;  }  findMedian(root->left, count, median, targetCount); // Traverse the left subtree  (\*count)++; // Increment the count  if (\*count == targetCount) // If the current count matches the target count, update the median  {  \*median = root->data;  }  findMedian(root->right, count, median, targetCount); // Traverse the right subtree  }  int countNodes(Node\* root) // Function to calculate the number of nodes in the BST  {  if (root == NULL)  {  return 0;  }  return countNodes(root->left) + countNodes(root->right) + 1;  }  int findBSTMedian(Node\* root) // Function to find the median of the BST  {  int nodeCount = countNodes(root);  int count = 0;  int median = 0;  int targetCount = (nodeCount + 1) / 2; // Find the target count for the median  findMedian(root, &count, &median, targetCount); // Find the median using an inorder traversal  return median;  }  int main()  {  Node\* root = NULL; // Create an empty Binary Search Tree  int numNodes; // Input the number of nodes to be inserted  printf("Enter the number of nodes: ");  scanf("%d", &numNodes);  printf("Enter the nodes:\n"); // Input the nodes from the user  for (int i = 0; i < numNodes; i++)  {  int data;  scanf("%d", &data);  root = insertNode(root, data);  }  int median = findBSTMedian(root); // Find the median of the BST  printf("Median: %d\n", median); // Print the median  } |
| **Sample Input:** |
| Enter the number of nodes: 7  Enter the nodes: 12 1 87 98 2 5 9 |
| **Sample Output:** |
| Median: 9 |

|  |
| --- |
| **Approach:****POST-ORDER TRAVERSAL OF TREE DATA STRUCTURE** |
| **Problem statement** |
| Problem statement: *Simplifying Expressions*  Reason of approach: The reason why post-order traversal is used in this application is because it allows to evaluate the expression from the bottom up. This is because the operands are always evaluated before the operator in post-order traversal. This makes it easier to simplify expressions because we can evaluate the sub-expressions first and then combine them using the operators. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <ctype.h>  #include <string.h>  typedef struct Node  {  char data;  struct Node\* left;  struct Node\* right;  } Node;  Node\* createNode(char data)  {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (newNode == NULL)  {  printf("Memory allocation error!\n");  exit(1);  }  newNode->data = data;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  bool isOperator(char c)  {  return (c == '+' || c == '-' || c == '\*' || c == '/');  }  Node\* simplifyExpressionTree(Node\* root) {  if (root == NULL) {  return NULL;  }  root->left = simplifyExpressionTree(root->left);  root->right = simplifyExpressionTree(root->right);  if (isOperator(root->data) && !isOperator(root->left->data) && !isOperator(root->right->data)) {  int leftOperand = root->left->data - '0';  int rightOperand = root->right->data - '0';  int result;  switch (root->data) {  case '+':  result = leftOperand + rightOperand;  break;  case '-':  result = leftOperand - rightOperand;  break;  case '\*':  result = leftOperand \* rightOperand;  break;  case '/':  result = leftOperand / rightOperand;  break;  }  free(root->left);  free(root->right);  root->left = NULL;  root->right = NULL;  root->data = result + '0';  }  return root;  }  Node\* buildExpressionTree(char\* postfix) {  Node\* stack[100];  int top = -1;  for (int i = 0; postfix[i] != '\0'; i++) {  if (isdigit(postfix[i])) {  Node\* newNode = createNode(postfix[i]);  stack[++top] = newNode;  } else if (isOperator(postfix[i])) {  Node\* newNode = createNode(postfix[i]);  newNode->right = stack[top--];  newNode->left = stack[top--];  stack[++top] = newNode;  }  }  return stack[top];  }  int evaluateExpressionTree(Node\* root) {  if (root == NULL) {  return 0;  }  if (!isOperator(root->data)) {  return root->data - '0';  }  int leftOperand = evaluateExpressionTree(root->left);  int rightOperand = evaluateExpressionTree(root->right);  int result;  switch (root->data) {  case '+':  result = leftOperand + rightOperand;  break;  case '-':  result = leftOperand - rightOperand;  break;  case '\*':  result = leftOperand \* rightOperand;  break;  case '/':  result = leftOperand / rightOperand;  break;  }  return result;  }  int main() {  char postfix[100];  printf("Enter the postfix expression: ");  fgets(postfix, sizeof(postfix), stdin);  Node\* root = buildExpressionTree(postfix);  root = simplifyExpressionTree(root);  int result = evaluateExpressionTree(root);  printf("Result: %d\n", result);  } |
| **Sample Input:** |
| Enter the postfix expression: 34\*2+ |
| **Sample Output:** |
| Result: 14 |

|  |
| --- |
| **Approach: DFS OF GRAPHS** |
| **Problem statement** |
| Problem statement: *Detecting Cycles in Undirected Graphs - Given an undirected graph, determine if it contains any cycles.*  Reason of approach: Depth-first search (DFS) is used to detect cycles in undirected graphs because it provides guarantees that breadth-first search (BFS) does not. In an undirected graph, a cycle exists whenever there are two paths between any pair of vertices. This is easy to detect during either BFS or DFS – The edges traced to new vertices form a tree, and any other edge indicates a cycle. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define MAX\_VERTICES 1000  typedef struct Node // Structure for representing an adjacency list node  {  int vertex;  struct Node\* next;  } Node;  typedef struct Graph // Structure for representing the undirected graph  {  int numVertices;  Node\* adjacencyList[MAX\_VERTICES];  } Graph;  Node\* createNode(int vertex) // Function to create a new adjacency list node  {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (newNode == NULL)  {  printf("Memory allocation error!\n");  exit(1);  }  newNode->vertex = vertex;  newNode->next = NULL;  return newNode;  }  void addEdge(Graph\* graph, int src, int dest) // Function to add an edge to the undirected graph  {  Node\* newNode = createNode(dest); // Add an edge from source to destination  newNode->next = graph->adjacencyList[src];  graph->adjacencyList[src] = newNode;  newNode = createNode(src); // Add an edge from destination to source  newNode->next = graph->adjacencyList[dest];  graph->adjacencyList[dest] = newNode;  }  bool isCyclicUtil(Graph\* graph, int vertex, bool visited[], int parent) // Function to perform DFS and check for cycles  {  visited[vertex] = true;  Node\* current = graph->adjacencyList[vertex]; // Recursive traversal of adjacent vertices  while (current != NULL) {  int adjacentVertex = current->vertex;  if (!visited[adjacentVertex]) // If adjacent vertex is not visited, recursively call the function  {  if (isCyclicUtil(graph, adjacentVertex, visited, vertex))  return true;  }  // If adjacent vertex is visited and not the parent of the current vertex, a cycle exists  else if (adjacentVertex != parent)  {  return true;  }  current = current->next;  }  return false;  }  bool isCyclic(Graph\* graph) // Function to detect cycles in the undirected graph using DFS  {  bool visited[MAX\_VERTICES];  for (int i = 0; i < graph->numVertices; i++)  {  visited[i] = false;  }  for (int i = 0; i < graph->numVertices; i++) // Iterate through each vertex and perform DFS to check for cycles  {  if (!visited[i]) {  if (isCyclicUtil(graph, i, visited, -1))  return true;  }  }  return false;  }  int main()  {  Graph graph; // Create an undirected graph  graph.numVertices = 0;  int numVertices, numEdges; // Input the number of vertices and edges  printf("Enter the number of vertices: ");  scanf("%d", &numVertices);  printf("Enter the number of edges: ");  scanf("%d", &numEdges);  for (int i = 0; i < numVertices; i++) // Initialize adjacency list for each vertex  {  graph.adjacencyList[i] = NULL;  }  graph.numVertices = numVertices;  printf("Enter the edges (vertex1 vertex2):\n"); // Input the edges from the user  for (int i = 0; i < numEdges; i++) {  int vertex1, vertex2;  scanf("%d %d", &vertex1, &vertex2);  addEdge(&graph, vertex1, vertex2);  }  bool hasCycle = isCyclic(&graph); // Check if the graph contains any cycles  if (hasCycle) // Print the result  {  printf("The graph contains a cycle.\n");  } else {  printf("The graph does not contain any cycles.\n");  }  return 0;  } |
| **Sample Input:** |
| Enter the number of vertices: 5  Enter the number of edges: 4  Enter the edges (vertex1 vertex2):  1 2  2 4  4 7  2 1 |
| **Sample Output:** |
| The graph contains a cycle. |

|  |
| --- |
| **Approach: BFS OF GRAPHS** |
| **Problem statement** |
| Problem statement: *Navigation Systems and Routing Algorithms - Helping customers to find shortest path from source to destination so that they find Uber in a cost efficient way*:  Reason of approach: BFS is used for navigation systems and routing algorithms because it is more efficient than DFS in finding the shortest path between two nodes in an unweighted graph. In an unweighted graph, the shortest path is the path with the least number of edges. With BFS, we always reach a vertex from a given source using the minimum number of edges. |
| **Code** |
| #include <stdio.h>  #include <stdbool.h>  #include <limits.h>  #define MAX\_LOCATIONS 1000  typedef struct { // Structure to represent a location or node in the graph  int id;  char name[50];  // Additional attributes like coordinates, road connections, etc.  } Location;  typedef struct { // Structure to represent an edge connecting two locations  int source;  int destination;  int weight;  } Edge;  typedef struct { // Structure to represent the graph of locations and connections  Location locations[MAX\_LOCATIONS];  Edge connections[MAX\_LOCATIONS];  int numLocations;  int numConnections;  } Graph;  void initializeGraph(Graph\* graph) { / Function to initialize the graph with locations and connections  graph->numLocations = 6;  graph->numConnections = 7;  for (int i = 0; i < graph->numLocations; i++) { // Set location IDs and names  graph->locations[i].id = i;  sprintf(graph->locations[i].name, "Location%d", i);  }  // Set connections with weights  graph->connections[0].source = 0;  graph->connections[0].destination = 1;  graph->connections[0].weight = 3;  graph->connections[1].source = 0;  graph->connections[1].destination = 2;  graph->connections[1].weight = 2;  graph->connections[2].source = 1;  graph->connections[2].destination = 2;  graph->connections[2].weight = 5;  graph->connections[3].source = 1;  graph->connections[3].destination = 3;  graph->connections[3].weight = 2;  graph->connections[4].source = 2;  graph->connections[4].destination = 3;  graph->connections[4].weight = 1;  graph->connections[5].source = 2;  graph->connections[5].destination = 4;  graph->connections[5].weight = 6;  graph->connections[6].source = 3;  graph->connections[6].destination = 4;  graph->connections[6].weight = 4;  }  // Function to find the index of a location in the graph given its ID  int getLocationIndex(Graph\* graph, int locationId) {  // Iterate through the 'locations' array and find the index of the location with the given ID  // Return the index or -1 if not found  for (int i = 0; i < graph->numLocations; i++) {  if (graph->locations[i].id == locationId) {  return i;  }  }  return -1;  }  // Function to find the shortest path using Dijkstra's algorithm  void findShortestPath(Graph\* graph, int source, int destination) {  int distances[MAX\_LOCATIONS];  bool visited[MAX\_LOCATIONS];  int previous[MAX\_LOCATIONS];  int numLocations = graph->numLocations;  // Initialize distances, visited, and previous arrays  for (int i = 0; i < numLocations; i++) {  distances[i] = INT\_MAX;  visited[i] = false;  previous[i] = -1;  }  // Set the distance of the source location to 0  distances[source] = 0;  // Perform Dijkstra's algorithm  for (int i = 0; i < numLocations - 1; i++) {  // Find the location with the minimum distance from the set of unvisited locations  int minDistance = INT\_MAX;  int minIndex = -1;  for (int j = 0; j < numLocations; j++) {  if (!visited[j] && distances[j] < minDistance) {  minDistance = distances[j];  minIndex = j;  }  }  // Mark the selected location as visited  visited[minIndex] = true;  // Update the distances of adjacent locations  for (int j = 0; j < graph->numConnections; j++) {  if (graph->connections[j].source == minIndex) {  int neighbor = graph->connections[j].destination;  int weight = graph->connections[j].weight;  if (distances[minIndex] + weight < distances[neighbor]) {  distances[neighbor] = distances[minIndex] + weight;  previous[neighbor] = minIndex;  }  }  }  }  // Print the shortest path and its distance  printf("Shortest Path from %s to %s:\n", graph->locations[source].name, graph->locations[destination].name);  int current = destination;  printf("%s", graph->locations[current].name);  while (previous[current] != -1) {  printf(" <- %s", graph->locations[previous[current]].name);  current = previous[current];  }  printf("\nDistance: %d\n", distances[destination]);  }  int main() {  Graph graph;  // Initialize the graph with locations and connections  initializeGraph(&graph);  // Get the source and destination locations from the user  int source, destination;  printf("Enter the source location ID: ");  scanf("%d", &source);  printf("Enter the destination location ID: ");  scanf("%d", &destination);  // Find the shortest path using Dijkstra's algorithm  findShortestPath(&graph, source, destination);  return 0;  } |
| **Sample Input:** |
| Enter the source location ID: 0  Enter the destination location ID: 4 |
| **Sample Output:** |
| Shortest Path from Location0 to Location4: Location4 <- Location3 <- Location2 <- Location0  Distance: 7 |

|  |
| --- |
| **Approach: LINEAR PROBING OF HASHING** |
| **Problem statement** |
| Problem statement: *Efficient Insertion and Retrieval - Implementing a simple hashing system using separate chaining to efficiently insert and retrieve data elements*  Reason of approach: Linear probing is used for efficient insertion and retrieval in hashing because it is one of many algorithms designed to find the correct position of a key in a hash table. When inserting keys, we mitigate collisions by scanning the cells in the table sequentially and when the next available cell is found, the key is inserted. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #define TABLE\_SIZE 100  // Structure for the data element  typedef struct {  int key;  int value;  } DataElement;  // Structure for the node in the hash table  typedef struct Node {  DataElement data;  struct Node\* next;  } Node;  // Structure for the hash table  typedef struct {  Node\* buckets[TABLE\_SIZE];  } HashTable;  // Function to create a new node for the hash table  Node\* createNode(DataElement data) {  Node\* newNode = (Node\*)malloc(sizeof(Node));  if (newNode == NULL) {  printf("Memory allocation error!\n");  exit(1);  }  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to initialize the hash table  void initializeHashTable(HashTable\* hashTable) {  for (int i = 0; i < TABLE\_SIZE; i++) {  hashTable->buckets[i] = NULL;  }  }  // Function to compute the hash value using a simple modulo function  int hashFunction(int key) {  return key % TABLE\_SIZE;  }  // Function to insert a data element into the hash table  void insert(HashTable\* hashTable, DataElement data) {  int hashValue = hashFunction(data.key);  Node\* newNode = createNode(data);  if (hashTable->buckets[hashValue] == NULL) {  // If the bucket is empty, insert the node as the first element  hashTable->buckets[hashValue] = newNode;  } else {  // If the bucket is not empty, insert the node at the beginning of the linked list  newNode->next = hashTable->buckets[hashValue];  hashTable->buckets[hashValue] = newNode;  }  }  // Function to retrieve a data element from the hash table based on its key  DataElement retrieve(HashTable\* hashTable, int key) {  int hashValue = hashFunction(key);  Node\* currentNode = hashTable->buckets[hashValue];  while (currentNode != NULL) {  if (currentNode->data.key == key) {  // If the key matches, return the data element  return currentNode->data;  }  currentNode = currentNode->next;  }  // If the key is not found, return an empty data element  DataElement emptyData = {0, 0};  return emptyData;  }  int main() {  HashTable hashTable;  initializeHashTable(&hashTable);  // Get the number of data elements from the user  int numElements;  printf("Enter the number of data elements: ");  scanf("%d", &numElements);  // Get the data elements from the user and insert them into the hash table  for (int i = 0; i < numElements; i++) {  DataElement data;  printf("Enter the key and value for element %d: ", i + 1);  scanf("%d %d", &data.key, &data.value);  insert(&hashTable, data);  }  // Get the key of the data element to retrieve from the user  int keyToRetrieve;  printf("Enter the key of the data element to retrieve: ");  scanf("%d", &keyToRetrieve);  // Retrieve the data element from the hash table  DataElement retrievedData = retrieve(&hashTable, keyToRetrieve);  // Print the retrieved data element  if (retrievedData.key != 0) {  printf("Retrieved Data: Key=%d, Value=%d\n", retrievedData.key, retrievedData.value);  } else {  printf("Data element not found!\n");  }  return 0;  } |
| **Sample Input:** |
| Enter the number of data elements: 3  Enter the key and value for element 1: 10 100  Enter the key and value for element 2: 25 250  Enter the key and value for element 3: 37 370  Enter the key of the data element to retrieve: 25 |
| **Sample Output:** |
| Retrieved Data: Key=25, Value=250 |

|  |
| --- |
| **Approach:** **QUADRATIC PROBING OF HASHING** |
| **Problem statement** |
| Problem statement: *Compiler Symbol Tables - Designing a compiler symbol table system for real-life programming languages. The goal is to develop a data structure and operations that efficiently handle symbol table management during the compilation process.*  Reason of approach: Quadratic probing of hashing is used for compiler symbol tables because it is a collision resolution technique that is more efficient than linear probing. It is also known to be faster than separate chaining. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define TABLE\_SIZE 100  // Structure for symbol entry in the symbol table  typedef struct {  char identifier[50];  char type[20];  // Additional attributes like scope, visibility, etc.  } SymbolEntry;  // Structure for the symbol table  typedef struct {  SymbolEntry\* entries[TABLE\_SIZE];  } SymbolTable;  // Function to create a new symbol entry  SymbolEntry\* createSymbolEntry(const char\* identifier, const char\* type) {  SymbolEntry\* entry = (SymbolEntry\*)malloc(sizeof(SymbolEntry));  if (entry == NULL) {  printf("Memory allocation error!\n");  exit(1);  }  strcpy(entry->identifier, identifier);  strcpy(entry->type, type);  return entry;  }  // Function to initialize the symbol table  void initializeSymbolTable(SymbolTable\* symbolTable) {  for (int i = 0; i < TABLE\_SIZE; i++) {  symbolTable->entries[i] = NULL;  }  }  // Function to compute the hash value for an identifier  int hashFunction(const char\* identifier) {  int sum = 0;  for (int i = 0; identifier[i] != '\0'; i++) {  sum += identifier[i];  }  return sum % TABLE\_SIZE;  }  // Function to insert a symbol entry into the symbol table  void insertSymbol(SymbolTable\* symbolTable, SymbolEntry\* entry) {  int hashValue = hashFunction(entry->identifier);  symbolTable->entries[hashValue] = entry;  }  // Function to retrieve a symbol entry from the symbol table  SymbolEntry\* retrieveSymbol(SymbolTable\* symbolTable, const char\* identifier) {  int hashValue = hashFunction(identifier);  return symbolTable->entries[hashValue];  }  int main() {  SymbolTable symbolTable;  initializeSymbolTable(&symbolTable);  // Get the number of symbol entries from the user  int numEntries;  printf("Enter the number of symbol entries: ");  scanf("%d", &numEntries);  // Get the symbol entries from the user and insert them into the symbol table  for (int i = 0; i < numEntries; i++) {  char identifier[50];  char type[20];  printf("Enter the identifier for entry %d: ", i + 1);  scanf("%s", identifier);  printf("Enter the type for entry %d: ", i + 1);  scanf("%s", type);  SymbolEntry\* entry = createSymbolEntry(identifier, type);  insertSymbol(&symbolTable, entry);  }  // Get the identifier of the symbol entry to retrieve from the user  char identifierToRetrieve[50];  printf("Enter the identifier of the symbol entry to retrieve: ");  scanf("%s", identifierToRetrieve);  // Retrieve the symbol entry from the symbol table  SymbolEntry\* retrievedEntry = retrieveSymbol(&symbolTable, identifierToRetrieve);  // Print the retrieved symbol entry  if (retrievedEntry != NULL) {  printf("Retrieved Symbol Entry: Identifier=%s, Type=%s\n", retrievedEntry->identifier, retrievedEntry->type);  } else {  printf("Symbol entry not found!\n");  }  return 0;  } |
| **fSample Input:** |
| Enter the number of symbol entries: 4  Enter the identifier for entry 1: w  Enter the type for entry 1: int  Enter the identifier for entry 2: x  Enter the type for entry 2: float  Enter the identifier for entry 3: y  Enter the type for entry 3: int  Enter the identifier for entry 4: z  Enter the type for entry 4: char  Enter the identifier of the symbol entry to retrieve: x |
| **Sample Output:** |
| Retrieved Symbol Entry: Identifier=x, Type=float |

|  |
| --- |
| **Approach: DOUBLE HASHING** |
| **Problem statement** |
| Problem statement: *Open Addressing Hash Tables - Design a phone directory system that utilizes double hashing to efficiently store and retrieve phone numbers and associated contact information. The system should provide fast access times, handle collisions, and allow users to easily search and update phone records.*  Reason of approach: Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing that is why it is used in open addressing hash tables. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define TABLE\_SIZE 100  // Structure for phone record in the phone directory  typedef struct {  char phoneNumber[15];  char name[50];  char address[100];  char email[50];  // Additional contact information  } PhoneRecord;  // Structure for the phone directory  typedef struct {  PhoneRecord\* entries[TABLE\_SIZE];  } PhoneDirectory;  // Function to create a new phone record  PhoneRecord\* createPhoneRecord(const char\* phoneNumber, const char\* name, const char\* address, const char\* email) {  PhoneRecord\* record = (PhoneRecord\*)malloc(sizeof(PhoneRecord));  if (record == NULL) {  printf("Memory allocation error!\n");  exit(1);  }  strcpy(record->phoneNumber, phoneNumber);  strcpy(record->name, name);  strcpy(record->address, address);  strcpy(record->email, email);  return record;  }  // Function to initialize the phone directory  void initializePhoneDirectory(PhoneDirectory\* phoneDirectory) {  for (int i = 0; i < TABLE\_SIZE; i++) {  phoneDirectory->entries[i] = NULL;  }  }  // Function to compute the primary hash value for a phone number  int primaryHashFunction(const char\* phoneNumber) {  int sum = 0;  for (int i = 0; phoneNumber[i] != '\0'; i++) {  sum += phoneNumber[i];  }  return sum % TABLE\_SIZE;  }  // Function to compute the secondary hash value for a phone number  int secondaryHashFunction(const char\* phoneNumber) {  // Choose a secondary hash function suitable for phone numbers  // Example: Multiply each digit of the phone number and take modulo TABLE\_SIZE  int product = 1;  for (int i = 0; phoneNumber[i] != '\0'; i++) {  if (phoneNumber[i] != '-')  product \*= phoneNumber[i] - '0';  }  return product % TABLE\_SIZE;  }  // Function to insert a phone record into the phone directory  void insertPhoneRecord(PhoneDirectory\* phoneDirectory, PhoneRecord\* record) {  int primaryHash = primaryHashFunction(record->phoneNumber);  int secondaryHash = secondaryHashFunction(record->phoneNumber);  int index = primaryHash;  int step = secondaryHash;  int i = 0;  // Double hashing collision resolution  while (phoneDirectory->entries[index] != NULL) {  index = (primaryHash + i \* step) % TABLE\_SIZE;  i++;  }  phoneDirectory->entries[index] = record;  }  // Function to retrieve a phone record from the phone directory based on a phone number  PhoneRecord\* retrievePhoneRecord(PhoneDirectory\* phoneDirectory, const char\* phoneNumber) {  int primaryHash = primaryHashFunction(phoneNumber);  int secondaryHash = secondaryHashFunction(phoneNumber);  int index = primaryHash;  int step = secondaryHash;  int i = 0;  // Double hashing collision resolution  while (phoneDirectory->entries[index] != NULL && strcmp(phoneDirectory->entries[index]->phoneNumber, phoneNumber) != 0) {  index = (primaryHash + i \* step) % TABLE\_SIZE;  i++;  }  return phoneDirectory->entries[index];  }  int main() {  PhoneDirectory phoneDirectory;  initializePhoneDirectory(&phoneDirectory);  int numRecords;  printf("Enter the number of phone records to insert: ");  scanf("%d", &numRecords);  for (int i = 0; i < numRecords; i++) {  char phoneNumber[15];  char name[50];  char address[100];  char email[50];  printf("Enter the phone number: ");  scanf("%s", phoneNumber);  printf("Enter the name: ");  scanf("%s", name);  printf("Enter the address: ");  scanf("%s", address);  printf("Enter the email: ");  scanf("%s", email);  PhoneRecord\* record = createPhoneRecord(phoneNumber, name, address, email);  insertPhoneRecord(&phoneDirectory, record);  }  char phoneNumberToRetrieve[15];  printf("Enter the phone number to retrieve: ");  scanf("%s", phoneNumberToRetrieve);  PhoneRecord\* retrievedRecord = retrievePhoneRecord(&phoneDirectory, phoneNumberToRetrieve);  if (retrievedRecord != NULL) {  printf("Retrieved Phone Record:\n");  printf("Phone Number: %s\n", retrievedRecord->phoneNumber);  printf("Name: %s\n", retrievedRecord->name);  printf("Address: %s\n", retrievedRecord->address);  printf("Email: %s\n", retrievedRecord->email);  } else {  printf("Phone record not found!\n");  }  return 0;  } |
| **Sample Input:** |
| Enter the number of phone records to insert: 3  Enter the phone number: 4554533221  Enter the name: Eric  Enter the address: Poland  Enter the email: erictesher@gmail.com  Enter the phone number: 4567899876  Enter the name: Warner  Enter the address: NYC  Enter the email: warnerbros@gmail.com  Enter the phone number: 098765432  Enter the name: Ashley  Enter the address: Ireland  Enter the email: ashleydavid@yahoo.com  Enter the phone number to retrieve: 098765432 |
| **Sample Output:** |
| Retrieved Phone Record:  Phone Number: 098765432  Name: Ashley  Address: Ireland  Email: ashleydavid@yahoo.com |