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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | **02** | | |  |  | | | |
| **Student Name** | Vinaya Shetti | | |  |  | | | |
| **SRN** | 01FE21BEC232 | | **Roll Number** | | 452 | **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 Parsing**: Pre-order traversal is utilized to convert mathematical expressions from infix notation (standard notation) to prefix notation (Polish notation) or postfix notation (reverse Polish notation). This is commonly used in compilers and calculator applications. | | | | | | | |
| 2. **Binary Search Trees (BST)**: Pre-order traversal is useful for traversing the entire BST to retrieve elements in ascending order. It also helps in reconstructing the original BST from the pre-order traversal sequence. | | | | | | | |
| **In-order traversal of tree data structure** | 1. **Finding Kth Smallest Element**: In-order traversal can efficiently find the Kth smallest element in a binary search tree. | | | | | | | |
| 2. **Expression Evaluation**: In-order traversal can be used to evaluate expressions stored in binary expression trees. By traversing the tree in-order, we can ensure that the correct order of operations is followed. | | | | | | | |
| **Post-order traversal of tree data structure** | 1. **Deleting a Binary Tree** When deallocating a binary tree, post-order traversal ensures that nodes are deleted in the correct order. Starting from the leaf nodes and moving upwards, the traversal deallocates memory for each node, eventually freeing the entire tree. | | | | | | | |
| 2. **File System Operations**: In file systems with hierarchical structures, post-order traversal can be used to perform actions such as deleting directories and files. This ensures that files and directories are deleted from the leaves to the root, maintaining consistency. | | | | | | | |
| **DFS of graphs** | 1. **Cycle Detection**: DFS can detect cycles in a graph. When exploring the graph, if we encounter an already visited node (back edge), it indicates the presence of a cycle in the graph. | | | | | | | |
| 2. **Finding Bridges and Articulation Points**: DFS can identify bridges (edges whose removal increases the number of connected components) and articulation points (vertices whose removal increases the number of connected components). | | | | | | | |
| **BFS of graphs** | 1. **Shortest Path and Routing Algorithms**: BFS can be used to find the shortest path in an unweighted graph. It is commonly employed in network routing algorithms to find the shortest path between two nodes in a network. | | | | | | | |
| 2. **Web Crawling**: BFS is used in web crawling and indexing applications to traverse and discover web pages on the internet. It helps in exploring the web in a systematic manner by visiting all pages at a given depth before moving to deeper levels. | | | | | | | |
| **Linear probing of hashing** | 1. **Hash Table Implementation**: Linear probing is commonly used in implementing hash tables for general-purpose data storage. It efficiently resolves collisions, and it's relatively simple to implement compared to other collision resolution techniques. | | | | | | | |
| 2. **Caching and Memory Management**: In systems that use hash tables for caching or memory management purposes, linear probing can be an efficient choice for handling collisions. It allows for quick retrieval and updates of cached data or memory allocations. | | | | | | | |
| **Quadratic probing of hashing** | 1. **Database Indexing**: Quadratic probing can be applied in database indexing structures, where it ensures that index entries are evenly distributed across the hash table, reducing the risk of clustering. | | | | | | | |
| 2. **String Matching and Search**: Quadratic probing can be utilized in string matching and searching algorithms that use hash tables to index strings or substrings. | | | | | | | |
| **Double hashing** | 1. **Load Balancing**: Double hashing can be employed in load balancing algorithms to distribute data evenly across multiple servers or nodes. It helps ensure a uniform distribution of data, preventing hotspots and improving the overall system's performance. | | | | | | | |
| 2. | | | | | | | |

1

|  |
| --- |
| **Approach:** Pre-order traversal of tree data structure |
| **Problem statement** |
| **Expression Parsing** The need for pre-order traversal arises from the nature of postfix expressions, where operators are placed after their corresponding operands. As we iterate through the postfix expression, we encounter operands first, and we need to create leaf nodes for them in the binary tree. However, operators, which determine the tree's structure, are encountered later. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  // Structure to represent a node in the binary expression tree  struct Node {  char data;  struct Node\* left;  struct Node\* right;  };  // Function to create a new node  struct Node\* createNode(char data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  // Function to build the binary expression tree from a postfix expression  struct Node\* buildExpressionTree(char postfix[]) {  struct Node\* stack[100];  int top = -1;  for (int i = 0; postfix[i]; i++) {  char symbol = postfix[i];  if (symbol >= '0' && symbol <= '9') {  stack[++top] = createNode(symbol);  } else {  struct Node\* rightOperand = stack[top--];  struct Node\* leftOperand = stack[top--];  struct Node\* operatorNode = createNode(symbol);  operatorNode->left = leftOperand;  operatorNode->right = rightOperand;  stack[++top] = operatorNode;  }  }  return stack[0];  }  // Function to perform pre-order traversal of the binary expression tree and evaluate the expression  int evaluateExpressionTree(struct Node\* root) {  if (root != NULL) {  if (root->data >= '0' && root->data <= '9') {  return root->data - '0'; // Convert char to integer  } else {  int leftResult = evaluateExpressionTree(root->left);  int rightResult = evaluateExpressionTree(root->right);  switch (root->data) {  case '+':  return leftResult + rightResult;  case '-':  return leftResult - rightResult;  case '\*':  return leftResult \* rightResult;  case '/':  return leftResult / rightResult;  default:  printf("Invalid operator.\n");  exit(EXIT\_FAILURE);  }  }  }  printf("Empty expression tree.\n");  exit(EXIT\_FAILURE);  }  // Function to perform pre-order traversal of the binary expression tree  void preOrderTraversal(struct Node\* root) {  if (root != NULL) {  printf("%c ", root->data);  preOrderTraversal(root->left);  preOrderTraversal(root->right);  }  }  int main() {  char postfixExpression[100];  printf("Enter the postfix expression: ");  scanf("%s", postfixExpression);  struct Node\* root = buildExpressionTree(postfixExpression);  printf("Pre-order expression: ");  preOrderTraversal(root);  printf("\n");  int result = evaluateExpressionTree(root);  printf("Result: %d\n", result);  return 0;  } |
| **Sample Input:** |
| Enter the postfix expression: 34+ |
| **Sample Output:** |
| Pre-order expression: + 3 4  Result: 7 |

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

2

|  |
| --- |
| **Approach:**  In-order traversal of tree data structure |
| **Problem statement** |
| **Finding Kth Smallest Element** In-order traversal is central to finding the Kth smallest element in a binary search tree. The process involves visiting nodes in ascending order, and by keeping track of the nodes visited, we can efficiently determine the Kth smallest element. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* left;  struct Node\* right;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  // Function to perform in-order traversal of the binary search tree  void inOrderTraversal(struct Node\* root) {  if (root != NULL) {  inOrderTraversal(root->left);  printf("%d ", root->data);  inOrderTraversal(root->right);  }  }  // Function to find the Kth smallest element in the binary search tree  int kthSmallestElement(struct Node\* root, int k) {  if (root == NULL || k <= 0)  return -1; // Invalid inputs or K out of range  struct Node\* stack[100];  int top = -1;  int count = 0;  struct Node\* current = root;  while (current != NULL || top != -1) {  while (current != NULL) {  stack[++top] = current;  current = current->left;  }  current = stack[top--];  count++;  if (count == k)  return current->data;  current = current->right;  }  return -1; // Kth element not found  }  int main() {  // Create a sample binary search tree  struct Node\* root = createNode(5);  root->left = createNode(3);  root->right = createNode(7);  root->left->left = createNode(2);  root->left->right = createNode(4);  root->right->left = createNode(6);  root->right->right = createNode(8);  printf("In-order traversal of the binary search tree: ");  inOrderTraversal(root);  printf("\n");  int k = 3;  int kthElement = kthSmallestElement(root, k);  if (kthElement != -1)  printf("The %dth smallest element is: %d\n", k, kthElement);  else  printf("The %dth smallest element does not exist.\n", k);  return 0;  } |
| **Sample Input:** |
| 5,3,7,2,4,6,8 |
| **Sample Output:** |
| The 4th smallest element is: 5 |

3

|  |
| --- |
| **Approach:** post-order traversal of tree data structure |
| **Problem statement** |
| **Deleting a Binary Tree**: By starting from the leaf nodes and moving upwards towards the root, post-order traversal ensures that memory for each node is deallocated before its parent node. This process continues until all nodes are deallocated, eventually freeing the entire tree. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* left;  struct Node\* right;  };  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  // Function to deallocate memory for the binary tree using post-order traversal  void deallocateTree(struct Node\* root) {  if (root != NULL) {  // Deallocate memory for left subtree  deallocateTree(root->left);  // Deallocate memory for right subtree  deallocateTree(root->right);  // Deallocate memory for current node  free(root);  }  }  int main() {  // Create a sample binary tree  struct Node\* root = createNode(1);  root->left = createNode(2);  root->right = createNode(3);  root->left->left = createNode(4);  root->left->right = createNode(5);  root->right->left = createNode(6);  root->right->right = createNode(7);  // Deallocate memory for the binary tree using post-order traversal  deallocateTree(root);  return 0;  } |
| **Sample Input:** |
|  |
| **Sample Output:** |
|  |

4

|  |
| --- |
| **Approach:** DFS |
| **Problem statement** |
| **Cycle Detection** By efficiently detecting cycles using DFS, we can validate the graph's structure, avoid unwanted behaviour, and improve the efficiency of various graph-related algorithms and applications. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_VERTICES 100  struct Node {  int vertex;  struct Node\* next;  };  struct Graph {  struct Node\* adjList[MAX\_VERTICES];  int visited[MAX\_VERTICES];  };  // Function to create a new node  struct Node\* createNode(int v) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->vertex = v;  newNode->next = NULL;  return newNode;  }  // Function to create a graph with 'V' vertices  struct Graph\* createGraph(int V) {  struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));  for (int i = 0; i < V; i++) {  graph->adjList[i] = NULL;  graph->visited[i] = 0;  }  return graph;  }  // Function to add an edge to an undirected graph  void addEdge(struct Graph\* graph, int src, int dest) {  // Add an edge from src to dest  struct Node\* newNode = createNode(dest);  newNode->next = graph->adjList[src];  graph->adjList[src] = newNode;  // Add an edge from dest to src (since it's an undirected graph)  newNode = createNode(src);  newNode->next = graph->adjList[dest];  graph->adjList[dest] = newNode;  }  // Recursive function to detect cycles in an undirected graph  int isCyclicUtil(struct Graph\* graph, int v, int parent) {  graph->visited[v] = 1;  // Explore all adjacent vertices of v  struct Node\* current = graph->adjList[v];  while (current != NULL) {  int adjVertex = current->vertex;  if (!graph->visited[adjVertex]) {  // If an adjacent vertex is not visited, recursively check for cycles  if (isCyclicUtil(graph, adjVertex, v))  return 1;  } else if (adjVertex != parent) {  // If an adjacent vertex is visited and not the parent of the current vertex,  // then a cycle exists  return 1;  }  current = current->next;  }  return 0;  }  // Function to detect cycles in an undirected graph  int isCyclic(struct Graph\* graph, int V) {  for (int i = 0; i < V; i++) {  if (!graph->visited[i]) {  // Call the recursive function for all unvisited vertices  if (isCyclicUtil(graph, i, -1))  return 1;  }  }  return 0;  }  int main() {  int V = 4; // Number of vertices in the graph  struct Graph\* graph = createGraph(V);  // Add edges to the graph to create a cycle  addEdge(graph, 0, 1);  addEdge(graph, 1, 2);  addEdge(graph, 2, 3);  addEdge(graph, 3, 0);  // Check if the graph contains a cycle  if (isCyclic(graph, V))  printf("The graph contains a cycle.\n");  else  printf("The graph does not contain a cycle.\n");  return 0;  } |
| **Sample Input:** |
| *<Sample Input>* |
| **Sample Output:** |
| *<Sample Output>* |

5

|  |
| --- |
| **Approach:** BFS |
| **Problem statement** |
| **Shortest Path and Routing Algorithms** By leveraging BFS's breadth-first exploration, the algorithm guarantees that the shortest path to each node is found first, making it a preferred choice for finding shortest paths in unweighted graphs. In network routing algorithms, this ensures efficient and optimal data delivery between network nodes, reducing latency and improving overall network performance. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_VERTICES 100  struct Node {  int vertex;  struct Node\* next;  };  struct Graph {  struct Node\* adjList[MAX\_VERTICES];  int visited[MAX\_VERTICES];  int distance[MAX\_VERTICES];  };  // Function to create a new node  struct Node\* createNode(int v) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->vertex = v;  newNode->next = NULL;  return newNode;  }  // Function to create a graph with 'V' vertices  struct Graph\* createGraph(int V) {  struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));  for (int i = 0; i < V; i++) {  graph->adjList[i] = NULL;  graph->visited[i] = 0;  graph->distance[i] = -1; // Initialize distances to -1 (indicating infinity)  }  return graph;  }  // Function to add an edge to an undirected graph  void addEdge(struct Graph\* graph, int src, int dest) {  // Add an edge from src to dest  struct Node\* newNode = createNode(dest);  newNode->next = graph->adjList[src];  graph->adjList[src] = newNode;  // Add an edge from dest to src (since it's an undirected graph)  newNode = createNode(src);  newNode->next = graph->adjList[dest];  graph->adjList[dest] = newNode;  }  // Function to perform BFS traversal and find the shortest path  void BFS(struct Graph\* graph, int start) {  int queue[MAX\_VERTICES];  int front = 0;  int rear = 0;  // Enqueue the starting vertex and mark it as visited  queue[rear++] = start;  graph->visited[start] = 1;  graph->distance[start] = 0;  while (front < rear) {  int currentVertex = queue[front++];  struct Node\* temp = graph->adjList[currentVertex];  while (temp != NULL) {  int adjVertex = temp->vertex;  if (!graph->visited[adjVertex]) {  // Enqueue the unvisited neighbor and mark it as visited  queue[rear++] = adjVertex;  graph->visited[adjVertex] = 1;  graph->distance[adjVertex] = graph->distance[currentVertex] + 1;  }  temp = temp->next;  }  }  }  int main() {  int V = 6; // Number of vertices in the graph  struct Graph\* graph = createGraph(V);  // Add edges to the graph  addEdge(graph, 0, 1);  addEdge(graph, 0, 2);  addEdge(graph, 1, 2);  addEdge(graph, 1, 3);  addEdge(graph, 2, 3);  addEdge(graph, 3, 4);  addEdge(graph, 4, 5);  int startVertex = 0; // Starting vertex for BFS  // Perform BFS from the starting vertex  BFS(graph, startVertex);  // Print the shortest distances from the starting vertex to all other vertices  printf("Shortest distances from vertex %d:\n", startVertex);  for (int i = 0; i < V; i++) {  printf("Vertex %d: %d\n", i, graph->distance[i]);  }  return 0;  } |
| **Sample Input:** |
| *<Sample Input>* |
| **Sample Output:** |
| *<Sample Output>* |

6

|  |
| --- |
| **Approach:** Linear probing |
| **Problem statement** |
| **Hash Table Implementation** linear probing remains a popular choice for hash table implementations, especially in scenarios where simplicity, low memory, and cache-friendly behaviour are desired. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_VERTICES 100  struct Node {  int vertex;  struct Node\* next;  };  struct Graph {  struct Node\* adjList[MAX\_VERTICES];  int visited[MAX\_VERTICES];  int distance[MAX\_VERTICES];  };  // Function to create a new node  struct Node\* createNode(int v) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->vertex = v;  newNode->next = NULL;  return newNode;  }  // Function to create a graph with 'V' vertices  struct Graph\* createGraph(int V) {  struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));  for (int i = 0; i < V; i++) {  graph->adjList[i] = NULL;  graph->visited[i] = 0;  graph->distance[i] = -1; // Initialize distances to -1 (indicating infinity)  }  return graph;  }  // Function to add an edge to an undirected graph  void addEdge(struct Graph\* graph, int src, int dest) {  // Add an edge from src to dest  struct Node\* newNode = createNode(dest);  newNode->next = graph->adjList[src];  graph->adjList[src] = newNode;  // Add an edge from dest to src (since it's an undirected graph)  newNode = createNode(src);  newNode->next = graph->adjList[dest];  graph->adjList[dest] = newNode;  }  // Function to perform BFS traversal and find the shortest path  void BFS(struct Graph\* graph, int start) {  int queue[MAX\_VERTICES];  int front = 0;  int rear = 0;  // Enqueue the starting vertex and mark it as visited  queue[rear++] = start;  graph->visited[start] = 1;  graph->distance[start] = 0;  while (front < rear) {  int currentVertex = queue[front++];  struct Node\* temp = graph->adjList[currentVertex];  while (temp != NULL) {  int adjVertex = temp->vertex;  if (!graph->visited[adjVertex]) {  // Enqueue the unvisited neighbor and mark it as visited  queue[rear++] = adjVertex;  graph->visited[adjVertex] = 1;  graph->distance[adjVertex] = graph->distance[currentVertex] + 1;  }  temp = temp->next;  }  }  }  int main() {  int V = 6; // Number of vertices in the graph  struct Graph\* graph = createGraph(V);  // Add edges to the graph  addEdge(graph, 0, 1);  addEdge(graph, 0, 2);  addEdge(graph, 1, 2);  addEdge(graph, 1, 3);  addEdge(graph, 2, 3);  addEdge(graph, 3, 4);  addEdge(graph, 4, 5);  int startVertex = 0; // Starting vertex for BFS  // Perform BFS from the starting vertex  BFS(graph, startVertex);  // Print the shortest distances from the starting vertex to all other vertices  printf("Shortest distances from vertex %d:\n", startVertex);  for (int i = 0; i < V; i++) {  printf("Vertex %d: %d\n", i, graph->distance[i]);  }  return 0;  } |
| **Sample Input:** |
| *<Sample Input>* |
| **Sample Output:** |
| *<Sample Output>* |

7

|  |
| --- |
| **Approach:** Quadratic probing/double hashing |
| **Problem statement** |
| **Database Indexing** In database indexing, the goal is to create an index that allows for fast and efficient retrieval of records based on their keys. The index maps key values to the corresponding storage locations in the database. Hash tables are commonly used to implement such indexes, where the keys are hashed to generate unique hash codes, which are then used as indices in the hash table. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <string.h>  #define TABLE\_SIZE 10  struct StudentRecord {  int id;  char name[50];  int age;  };  struct IndexEntry {  int key; // Student ID  int recordIndex; // Index of the record in the database  };  struct HashTable {  struct IndexEntry table[TABLE\_SIZE];  };  // Function to initialize the hash table  void initializeHashTable(struct HashTable\* hashtable) {  for (int i = 0; i < TABLE\_SIZE; i++) {  hashtable->table[i].key = -1; // Initialize keys to -1 (indicating an empty slot)  hashtable->table[i].recordIndex = -1;  }  }  // Function to calculate the hash index using a simple hash function  int hashFunction(int key) {  return key % TABLE\_SIZE;  }  // Function to insert an index entry into the hash table using quadratic probing  void insert(struct HashTable\* hashtable, int key, int recordIndex) {  int index = hashFunction(key);  int i = 1;  while (hashtable->table[index].key != -1) {  // Quadratic probing: Move to the next slot using a quadratic function  index = (index + i \* i) % TABLE\_SIZE;  i++;  // If the entire table is probed or an empty slot is found, break the loop  if (i > TABLE\_SIZE || hashtable->table[index].key == -1) {  break;  }  }  // Insert the index entry in the empty slot  hashtable->table[index].key = key;  hashtable->table[index].recordIndex = recordIndex;  }  // Function to search for an index entry in the hash table  int search(struct HashTable\* hashtable, int key) {  int index = hashFunction(key);  int i = 1;  while (hashtable->table[index].key != key) {  // Quadratic probing: Move to the next slot using a quadratic function  index = (index + i \* i) % TABLE\_SIZE;  i++;  // If the entire table is probed or the key is not found, return -1  if (i > TABLE\_SIZE || hashtable->table[index].key == -1) {  return -1;  }  }  // Key found, return the index of the corresponding record in the database  return hashtable->table[index].recordIndex;  }  int main() {  struct HashTable hashtable;  initializeHashTable(&hashtable);  // Assume we have a database of student records  struct StudentRecord database[] = {  {101, "Alice", 20},  {102, "Bob", 21},  {103, "Charlie", 22},  // Add more records as needed  };  // Insert index entries into the hash table  for (int i = 0; i < sizeof(database) / sizeof(database[0]); i++) {  insert(&hashtable, database[i].id, i);  }  // Search for records based on student IDs  int searchId = 102;  int recordIndex = search(&hashtable, searchId);  if (recordIndex != -1) {  // Print the record details  printf("Record found for ID %d:\n", searchId);  printf("Name: %s\n", database[recordIndex].name);  printf("Age: %d\n", database[recordIndex].age);  } else {  printf("Record not found for ID %d.\n", searchId);  }  return 0;  } |
| **Sample Input:** |
| *<Sample Input>* |
| **Sample Output:** |
| *<Sample Output>* |

8

|  |
| --- |
| **Approach:** Double hashing |
| **Problem statement** |
| **Load Balancing** Load balancing algorithms with double hashing are widely used in various distributed systems, including web servers, content delivery networks (CDNs), cloud computing platforms, and distributed databases. By employing double hashing, load balancing algorithms can efficiently distribute requests and data |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <string.h>  #define NUM\_SERVERS 3  // Structure to represent data to be distributed  struct Data {  char key[20];  char value[50];  };  // Structure to represent a server  struct Server {  char name[20];  struct Data\* data;  int numData;  };  // Function to calculate the first hash index using hash function 1  int hashFunction1(char\* key, int numBuckets) {  int hash = 0;  for (int i = 0; key[i] != '\0'; i++) {  hash = (hash + key[i]) % numBuckets;  }  return hash;  }  // Function to calculate the second hash index using hash function 2  int hashFunction2(char\* key, int numBuckets) {  int hash = 0;  for (int i = 0; key[i] != '\0'; i++) {  hash = (hash + key[i] \* (i + 1)) % numBuckets;  }  return (2 \* hash + 1) % numBuckets; // Ensure an odd value to avoid clustering  }  // Function to add data to a server  void addToServer(struct Server\* server, char\* key, char\* value) {  server->data = realloc(server->data, (server->numData + 1) \* sizeof(struct Data));  strcpy(server->data[server->numData].key, key);  strcpy(server->data[server->numData].value, value);  server->numData++;  }  int main() {  // Create three servers  struct Server servers[NUM\_SERVERS];  for (int i = 0; i < NUM\_SERVERS; i++) {  sprintf(servers[i].name, "Server-%d", i + 1);  servers[i].data = NULL;  servers[i].numData = 0;  }  // Data to be distributed  struct Data dataToDistribute[] = {  {"key1", "value1"},  {"key2", "value2"},  {"key3", "value3"},  {"key4", "value4"},  {"key5", "value5"},  // Add more data as needed  };  int numDataToDistribute = sizeof(dataToDistribute) / sizeof(dataToDistribute[0]);  // Distribute data among servers using double hashing  for (int i = 0; i < numDataToDistribute; i++) {  int serverIndex = hashFunction1(dataToDistribute[i].key, NUM\_SERVERS);  int offset = hashFunction2(dataToDistribute[i].key, NUM\_SERVERS);  // Find an available server using double hashing  while (servers[serverIndex].data != NULL) {  serverIndex = (serverIndex + offset) % NUM\_SERVERS;  }  // Add the data to the chosen server  addToServer(&servers[serverIndex], dataToDistribute[i].key, dataToDistribute[i].value);  }  // Print the distributed data on each server  for (int i = 0; i < NUM\_SERVERS; i++) {  printf("Data on %s:\n", servers[i].name);  for (int j = 0; j < servers[i].numData; j++) {  printf("%s: %s\n", servers[i].data[j].key, servers[i].data[j].value);  }  printf("\n");  }  // Free memory  for (int i = 0; i < NUM\_SERVERS; i++) {  free(servers[i].data);  }  return 0;  } |
| **Sample Input:** |
| *<Sample Input>* |
| **Sample Output:** |
| *<Sample Output>* |