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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *02* | | |  |  | | | |
| **Student Name** | Ohil Hosamane | | |  |  | | | |
| **SRN** | 01FE21BEC358 | | **Roll Number** | | 408 | **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 | | | | | | | |
| 2. Directory/File Tree Listing | | | | | | | |
| **In-order traversal of tree data structure** | 1. Binary Search Tree (BST) Validation | | | | | | | |
| 2. Expression Tree Infix Conversion | | | | | | | |
| **Post-order traversal of tree data structure** | 1. Expression Tree Postfix Conversion | | | | | | | |
| 2. Memory Deallocation | | | | | | | |
| **DFS of graphs** | 1. Topological Sorting | | | | | | | |
| 2. Connected Components | | | | | | | |
| **BFS of graphs** | 1. Shortest Path | | | | | | | |
| 2. Web Crawling | | | | | | | |
| **Linear probing of hashing** | 1. Symbol Table | | | | | | | |
| 2. Caching | | | | | | | |
| **Quadratic probing of hashing** | 1. Spell Checking | | | | | | | |
| 2. Task Scheduling | | | | | | | |
| **Double hashing** | 1.Load Balancing | | | | | | | |
| 2. Sudoku Solving | | | | | | | |

|  |
| --- |
| **Approach:** *Pre-order traversal of tree data structure* |
| **Problem statement** |
| *Expression Tree Evaluation: An expression tree is a binary tree where each internal node represents an operator, and the leaf nodes represent operands. By performing a pre-order traversal of an expression tree, you can evaluate the expression and compute its result. This approach can be used in compilers, interpreters, or calculator applications.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *struct Node {*  *char data;*  *struct Node\* left;*  *struct Node\* right;*  *};*  *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;*  *}*  *struct Node\* buildExpressionTree() {*  *char c;*  *scanf(" %c", &c);*  *if (c == '\n' || c == ' ')*  *return NULL;*  *struct Node\* newNode = createNode(c);*  *newNode->left = buildExpressionTree();*  *newNode->right = buildExpressionTree();*  *return newNode;*  *}*  *int evaluateExpressionTree(struct Node\* root) {*  *if (root == NULL)*  *return 0;*  *if (root->left == NULL && root->right == NULL)*  *return root->data - '0';*  *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;*  *}*  *return 0;*  *}*  *void freeExpressionTree(struct Node\* root) {*  *if (root == NULL)*  *return;*  *freeExpressionTree(root->left);*  *freeExpressionTree(root->right);*  *free(root);*  *}*  *int main() {*  *printf("Enter the expression tree in pre-order traversal (use space for empty nodes):\n");*  *struct Node\* root = buildExpressionTree();*  *int result = evaluateExpressionTree(root);*  *printf("Result: %d\n", result);*  *freeExpressionTree(root);*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the expression tree in pre-order traversal (use space for empty nodes):*  *+ \* 2 3 - 6 1* |
| **Sample Output:** |
| *11* |

|  |
| --- |
| **Approach:** *In-order traversal of tree data structure* |
| **Problem statement** |
| *Binary Search Tree (BST) Validation: In-order traversal can be used to validate whether a given binary tree is a valid Binary Search Tree (BST) or not. By performing an in-order traversal, the elements of a BST will be visited in ascending order. If the elements are not in ascending order during the traversal, it indicates that the tree is not a valid BST.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <stdbool.h>*  *struct Node {*  *int data;*  *struct Node\* left;*  *struct Node\* right;*  *};*  *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;*  *}*  *struct Node\* buildBinaryTree() {*  *int data;*  *printf("Enter node data (or -1 for empty node): ");*  *scanf("%d", &data);*  *if (data == -1)*  *return NULL;*  *struct Node\* newNode = createNode(data);*  *printf("Enter left subtree of %d:\n", data);*  *newNode->left = buildBinaryTree();*  *printf("Enter right subtree of %d:\n", data);*  *newNode->right = buildBinaryTree();*  *return newNode;*  *}*  *bool validateBST(struct Node\* root, int\* prev) {*  *if (root == NULL)*  *return true;*  *if (!validateBST(root->left, prev))*  *return false;*  *if (root->data <= \*prev)*  *return false;*  *\*prev = root->data;*  *return validateBST(root->right, prev);*  *}*  *void freeBinaryTree(struct Node\* root) {*  *if (root == NULL)*  *return;*  *freeBinaryTree(root->left);*  *freeBinaryTree(root->right);*  *free(root);*  *}*  *int main() {*  *printf("Enter the binary tree:\n");*  *// Build the binary tree*  *struct Node\* root = buildBinaryTree();*  *int prev = -1;*  *bool isValidBST = validateBST(root, &prev);*  *if (isValidBST)*  *printf("The tree is a valid BST.\n");*  *else*  *printf("The tree is not a valid BST.\n");*  *freeBinaryTree(root);*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the binary tree:*  *Enter node data (or -1 for empty node): 4*  *Enter left subtree of 4:*  *Enter node data (or -1 for empty node): 2*  *Enter left subtree of 2:*  *Enter node data (or -1 for empty node): 1*  *Enter left subtree of 1:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 1:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 2:*  *Enter node data (or -1 for empty node): 3*  *Enter left subtree of 3:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 3:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 4:*  *Enter node data (or -1 for empty node): 5*  *Enter left subtree of 5:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 5:*  *Enter node data (or -1 for empty node): -1* |
| **Sample Output:** |
| *The tree is a valid BST.* |

|  |
| --- |
| **Approach:** *post-order traversal of tree data structure* |
| **Problem statement** |
| *Expression Tree Postfix Conversion: Post-order traversal can be used to convert an expression tree into its corresponding postfix notation. In an expression tree, the operators are stored at the internal nodes, and the operands are stored at the leaf nodes. By performing a post-order traversal, we can visit the nodes in the correct order and create the postfix expression.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *struct Node {*  *int data;*  *struct Node\* left;*  *struct Node\* right;*  *};*  *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;*  *}*  *struct Node\* buildBinaryTree() {*  *int data;*  *printf("Enter node data (or -1 for empty node): ");*  *scanf("%d", &data);*  *if (data == -1)*  *return NULL;*  *struct Node\* newNode = createNode(data);*  *printf("Enter left subtree of %d:\n", data);*  *newNode->left = buildBinaryTree();*  *printf("Enter right subtree of %d:\n", data);*  *newNode->right = buildBinaryTree();*  *return newNode;*  *}*  *void deallocateTree(struct Node\* root) {*  *if (root == NULL)*  *return;*  *deallocateTree(root->left);*  *deallocateTree(root->right);*  *free(root);*  *}*  *int main() {*  *printf("Enter the binary tree:\n");*  *struct Node\* root = buildBinaryTree();*  *deallocateTree(root);*  *printf("Memory deallocation complete.\n");*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the binary tree:*  *Enter node data (or -1 for empty node): 4*  *Enter left subtree of 4:*  *Enter node data (or -1 for empty node): 2*  *Enter left subtree of 2:*  *Enter node data (or -1 for empty node): 1*  *Enter left subtree of 1:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 1:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 2:*  *Enter node data (or -1 for empty node): 3*  *Enter left subtree of 3:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 3:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 4:*  *Enter node data (or -1 for empty node): 5*  *Enter left subtree of 5:*  *Enter node data (or -1 for empty node): -1*  *Enter right subtree of 5:*  *Enter node data (or -1 for empty node): -1* |
| **Sample Output:** |
| *Memory deallocation complete.* |

|  |
| --- |
| **Approach:** *DFS* |
| **Problem statement** |
| *Connected Components: DFS can be used to find and identify the connected components in an undirected graph. Connected components are subsets of vertices where each vertex is reachable from any other vertex within the same component. By performing DFS on each unvisited vertex, we can determine all the connected components in the graph. This information is useful for analyzing network connectivity, social network analysis, and graph clustering.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdbool.h>*  *#define MAX\_VERTICES 100*  *bool graph[MAX\_VERTICES][MAX\_VERTICES];*  *bool visited[MAX\_VERTICES];*  *int numVertices;*  *void initializeGraph() {*  *for (int i = 0; i < MAX\_VERTICES; i++) {*  *for (int j = 0; j < MAX\_VERTICES; j++) {*  *graph[i][j] = false;*  *}*  *visited[i] = false;*  *}*  *}*  *void addEdge(int u, int v) {*  *graph[u][v] = true;*  *graph[v][u] = true;*  *}*  *void dfs(int vertex) {*  *visited[vertex] = true;*  *printf("%d ", vertex);*  *for (int i = 0; i < numVertices; i++) {*  *if (graph[vertex][i] && !visited[i]) {*  *dfs(i);*  *}*  *}*  *}*  *void findConnectedComponents() {*  *for (int i = 0; i < numVertices; i++) {*  *if (!visited[i]) {*  *printf("Connected component: ");*  *dfs(i);*  *printf("\n");*  *}*  *}*  *}*  *int main() {*  *numVertices = 7;*  *initializeGraph();*  *addEdge(0, 1);*  *addEdge(0, 2);*  *addEdge(1, 2);*  *addEdge(3, 4);*  *addEdge(4, 5);*  *addEdge(5, 6);*  *printf("Connected Components:\n");*  *findConnectedComponents();*  *return 0;*  *}* |
| **Sample Input:** |
| *Vertices: 7*  *Edges: (0, 1), (0, 2), (1, 2), (3, 4), (4, 5), (5, 6)* |
| **Sample Output:** |
| *Connected component: 0 1 2*  *Connected component: 3 4 5 6* |

|  |
| --- |
| **Approach:** *BFS* |
| **Problem statement** |
| *Shortest Path: BFS can be used to find the shortest path between two vertices in an unweighted graph. It guarantees that the first path found between the source and destination vertices is the shortest path.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <stdbool.h>*  *#define MAX\_VERTICES 100*  *typedef struct Node {*  *int vertex;*  *struct Node\* next;*  *} Node;*  *Node\* createNode(int v) {*  *Node\* newNode = (Node\*)malloc(sizeof(Node));*  *newNode->vertex = v;*  *newNode->next = NULL;*  *return newNode;*  *}*  *typedef struct Graph {*  *Node\* adjList[MAX\_VERTICES];*  *bool visited[MAX\_VERTICES];*  *} Graph;*  *Graph\* createGraph() {*  *Graph\* graph = (Graph\*)malloc(sizeof(Graph));*  *int i;*  *for (i = 0; i < MAX\_VERTICES; i++) {*  *graph->adjList[i] = NULL;*  *graph->visited[i] = false;*  *}*  *return graph;*  *}*  *void addEdge(Graph\* graph, int src, int dest) {*  *Node\* newNode = createNode(dest);*  *newNode->next = graph->adjList[src];*  *graph->adjList[src] = newNode;*  *newNode = createNode(src);*  *newNode->next = graph->adjList[dest];*  *graph->adjList[dest] = newNode;*  *}*  *void BFS(Graph\* graph, int startVertex, int endVertex) {*  *int queue[MAX\_VERTICES];*  *int front = -1, rear = -1;*  *int parent[MAX\_VERTICES];*  *int distance[MAX\_VERTICES];*  *int i, currentVertex, adjacentVertex;*  *for (i = 0; i < MAX\_VERTICES; i++) {*  *parent[i] = -1;*  *distance[i] = -1;*  *}*  *graph->visited[startVertex] = true;*  *distance[startVertex] = 0;*  *queue[++rear] = startVertex;*  *while (front != rear) {*  *currentVertex = queue[++front];*  *Node\* temp = graph->adjList[currentVertex];*    *while (temp) {*  *adjacentVertex = temp->vertex;*  *if (!graph->visited[adjacentVertex]) {*  *queue[++rear] = adjacentVertex;*  *graph->visited[adjacentVertex] = true;*  *parent[adjacentVertex] = currentVertex;*  *distance[adjacentVertex] = distance[currentVertex] + 1;*  *if (adjacentVertex == endVertex) {*    *printf("Shortest path from %d to %d: ", startVertex, endVertex);*  *int path[MAX\_VERTICES];*  *int pathLength = 0;*  *int vertex = endVertex;*  *while (vertex != startVertex) {*  *path[pathLength++] = vertex;*  *vertex = parent[vertex];*  *}*  *printf("%d ", startVertex);*  *for (i = pathLength - 1; i >= 0; i--) {*  *printf("%d ", path[i]);*  *}*  *printf("\n");*  *return;*  *}*  *}*  *temp = temp->next;*  *}*  *}*    *printf("No path found from %d to %d\n", startVertex, endVertex);*  *}*  *int main() {*  *Graph\* graph = createGraph();*  *int numVertices, numEdges;*  *int startVertex, endVertex;*  *printf("Enter the number of vertices in the graph: ");*  *scanf("%d", &numVertices);*  *printf("Enter the number of edges in the graph: ");*  *scanf("%d", &numEdges);*  *printf("Enter the edges (source vertex, destination vertex):\n");*  *int i;*  *int src, dest;*  *for (i = 0; i < numEdges; i++) {*  *scanf("%d %d", &src, &dest);*  *addEdge(graph, src, dest);*  *}*  *printf("Enter the start vertex: ");*  *scanf("%d", &startVertex);*  *printf("Enter the end vertex: ");*  *scanf("%d", &endVertex);*  *BFS(graph, startVertex, endVertex);*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of vertices in the graph: 5*  *Enter the number of edges in the graph: 6*  *Enter the edges (source vertex, destination vertex):*  *0 1*  *0 2*  *1 3*  *2 3*  *2 4*  *3 4*  *Enter the start vertex: 0*  *Enter the end vertex: 4* |
| **Sample Output:** |
| *Shortest path from 0 to 4: 0 2 4* |

|  |
| --- |
| **Approach:** *Linear probing* |
| **Problem statement** |
| *Symbol Table: Linear probing can be used to implement a symbol table, which is a data structure that stores key-value pairs. Symbol tables are commonly used in programming languages to implement symbol lookup and storage, such as variable names and their associated values. Linear probing can efficiently handle collisions that occur when inserting or retrieving symbols from the table.* |
| **Code** |
| *#include <stdio.h>*  *#define SIZE 10*  *struct KeyValue {*  *int key;*  *int value;*  *};*  *struct KeyValue symbolTable[SIZE];*  *void insert(int key, int value) {*  *int hash = key % SIZE;*  *while (symbolTable[hash].key != -1) {*  *hash = (hash + 1) % SIZE;*  *}*    *symbolTable[hash].key = key;*  *symbolTable[hash].value = value;*  *}*  *int retrieve(int key) {*  *int hash = key % SIZE;*  *while (symbolTable[hash].key != key) {*  *hash = (hash + 1) % SIZE;*    *if (symbolTable[hash].key == -1) {*  *return -1;*  *}*  *}*  *return symbolTable[hash].value;*  *}*  *int main() {*    *for (int i = 0; i < SIZE; i++) {*  *symbolTable[i].key = -1;*  *symbolTable[i].value = -1;*  *}*  *int numPairs;*  *printf("Enter the number of key-value pairs: ");*  *scanf("%d", &numPairs);*  *printf("Enter the key-value pairs:\n");*  *for (int i = 0; i < numPairs; i++) {*  *int key, value;*  *printf("Key: ");*  *scanf("%d", &key);*  *printf("Value: ");*  *scanf("%d", &value);*  *insert(key, value);*  *}*  *printf("\nEnter the keys to retrieve their values (-1 to exit):\n");*  *int queryKey;*  *while (1) {*  *printf("Key: ");*  *scanf("%d", &queryKey);*  *if (queryKey == -1) {*  *break;*  *}*  *int value = retrieve(queryKey);*  *if (value == -1) {*  *printf("Key not found in the symbol table.\n");*  *} else {*  *printf("Value: %d\n", value);*  *}*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of key-value pairs: 3*  *Enter the key-value pairs:*  *Key: 10*  *Value: 100*  *Key: 20*  *Value: 200*  *Key: 30*  *Value: 300*  *Enter the keys to retrieve their values (-1 to exit):*  *Key: 20*  *Value: 200*  *Key: 40* |
| **Sample Output:** |
| *Key not found in the symbol table.*  *Key: -1* |

|  |
| --- |
| **Approach:** *quadratic probing* |
| **Problem statement** |
| *Spell Checking: Quadratic probing can be used to implement a spell-checking system. In this application, a hash table can be used to store a dictionary of valid words. When checking the spelling of a word, quadratic probing can be employed to handle collisions and find the next available slot in the hash table for the word being checked.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <string.h>*  *#define TABLE\_SIZE 10*  *struct Node {*  *char word[20];*  *};*  *struct Node symbolTable[TABLE\_SIZE];*  *int hashFunction(char \*word) {*  *int sum = 0;*  *for (int i = 0; word[i] != '\0'; i++) {*  *sum += word[i];*  *}*  *return sum % TABLE\_SIZE;*  *}*  *void insert(char \*word) {*  *int index = hashFunction(word);*  *int i = 1;*  *while (symbolTable[index].word[0] != '\0') {*  *index = (index + i \* i) % TABLE\_SIZE;*  *i++;*  *}*  *strcpy(symbolTable[index].word, word);*  *}*  *int search(char \*word) {*  *int index = hashFunction(word);*  *int i = 1;*  *while (symbolTable[index].word[0] != '\0') {*  *if (strcmp(symbolTable[index].word, word) == 0) {*  *return index;*  *}*  *index = (index + i \* i) % TABLE\_SIZE;*  *i++;*  *}*  *return -1;*  *}*  *int main() {*    *for (int i = 0; i < TABLE\_SIZE; i++) {*  *symbolTable[i].word[0] = '\0';*  *}*  *int numWords;*  *printf("Enter the number of words in the dictionary: ");*  *scanf("%d", &numWords);*  *printf("Enter the words:\n");*  *for (int i = 0; i < numWords; i++) {*  *char word[20];*  *printf("Word %d: ", i + 1);*  *scanf("%s", word);*  *insert(word);*  *}*  *int numQueries;*  *printf("\nEnter the number of words to search: ");*  *scanf("%d", &numQueries);*  *printf("Enter the words to search:\n");*  *for (int i = 0; i < numQueries; i++) {*  *char word[20];*  *printf("Word %d: ", i + 1);*  *scanf("%s", word);*  *int index = search(word);*  *if (index == -1) {*  *printf("Word '%s' is not found in the dictionary.\n", word);*  *} else {*  *printf("Word '%s' is found at index %d in the dictionary.\n", word, index);*  *}*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of words in the dictionary: 5*  *Enter the words:*  *Word 1: hello*  *Word 2: world*  *Word 3: open*  *Word 4: the*  *Word 5: model*  *Enter the number of words to search: 3*  *Enter the words to search:*  *Word 1: hello*  *Word 2: chat*  *Word 3: the* |
| **Sample Output:** |
| *Word 'hello' is found at index 6 in the dictionary.*  *Word 'chat' is not found in the dictionary.*  *Word 'the' is found at index 4 in the dictionary.* |

|  |
| --- |
| **Approach:** *Double hashing* |
| **Problem statement** |
| *Load Balancing: Double hashing can be applied in load balancing algorithms. Load balancers distribute incoming network traffic across multiple servers to optimize resource utilization and ensure high availability. Double hashing can help evenly distribute the incoming requests among the servers, preventing overload on any particular server and achieving better load balancing..* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define SERVER\_COUNT 5  struct Server {  int id;  int load;  };  struct Server serverList[SERVER\_COUNT];  int hashFunction1(int id) {  return id % SERVER\_COUNT;  }  int hashFunction2(int id) {  return 1 + (id % (SERVER\_COUNT - 1));  }  void initializeServers() {  for (int i = 0; i < SERVER\_COUNT; i++) {  serverList[i].id = i;  serverList[i].load = 0;  }  }  int getNextServer(int clientID) {  int index = hashFunction1(clientID);  int step = hashFunction2(clientID);  int i = 0;  while (serverList[index].load >= SERVER\_COUNT) {  index = (index + i \* step) % SERVER\_COUNT;  i++;  }  serverList[index].load++;  return serverList[index].id;  }  int main() {  int numClients;  printf("Enter the number of clients: ");  scanf("%d", &numClients);  initializeServers();  printf("Client ID\tAssigned Server ID\n");  for (int i = 0; i < numClients; i++) {  int clientID = i;  int serverID = getNextServer(clientID);  printf("%d\t\t%d\n", clientID, serverID);  }  return 0;  } |
| **Sample Input:** |
| *Enter the number of clients: 10*  *Client ID*  *0*  *1*  *2*  *3*  *4*  *5*  *6*  *7*  *8*  *9* |
| **Sample Output:** |
| *Assigned Server ID*  *0*  *1*  *2*  *3*  *4*  *0*  *1*  *2*  *3*  *4* |