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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | ***02*** | | |  |  | | | |
| **Student Name** | Prajwal B | | |  |  | | | |
| **SRN** | 01FE20BEC196 | | **Roll Number** | | 564 | **Division** | E | |
| **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 evaluation | | | | | | | |
| 2. Duplication of Trees | | | | | | | |
| **In-order traversal of tree data structure** | 1. It is used to get the BST nodes in increasing order. | | | | | | | |
| 2.To check validity of a binary search tree. | | | | | | | |
| **Post-order traversal of tree data structure** | 1. Memory deallocation in a tree | | | | | | | |
| 2. Expression evaluation in arithmetic expression trees | | | | | | | |
| **DFS of graphs** | 1. Finding connected components in an undirected graph. | | | | | | | |
| 2. Solving maze and pathfinding problems. | | | | | | | |
| **BFS of graphs** | 1. Finding the shortest path in an unweighted graph. | | | | | | | |
| 2. Web crawling or spidering to index web pages. | | | | | | | |
| **Linear probing of hashing** | 1. Implementing a hash table with linear probing. | | | | | | | |
| 2. Resizing a hash table with linear probing. | | | | | | | |
| **Quadratic probing of hashing** | 1. Implementing a hash table with quadratic probing for collision resolution. | | | | | | | |
| 2. Spell-checking a text document using a hash table with quadratic probing. | | | | | | | |
| **Double hashing** | 1. Implementing an Open Addressing Hash Table. | | | | | | | |
| 2. Detecting duplicate elements in an array using hash table. | | | | | | | |

|  |
| --- |
| **Approach:** *Pre-order traversal of tree data structure* |
| **Problem statement** |
| Pre-order traversal is particularly useful for expression parsing because it allows us to easily evaluate arithmetic expressions in prefix notation. In prefix notation, the operators appear before their operands, which makes it easier to parse using pre-order traversal compared to other traversal methods. |
| **Code** |
| ***#include <stdio.h>***  ***#include <stdlib.h>***  ***#include <ctype.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 = newNode->right = NULL;***  ***return newNode;***  ***}***  ***int evaluateExpression(struct Node\* root) {***  ***if (root == NULL) {***  ***return 0;***  ***}***  ***if (isdigit(root->data)) {***  ***return root->data - '0';***  ***}***  ***int leftValue = evaluateExpression(root->left);***  ***int rightValue = evaluateExpression(root->right);***  ***switch (root->data) {***  ***case '+':***  ***return leftValue + rightValue;***  ***case '-':***  ***return leftValue - rightValue;***  ***case '\*':***  ***return leftValue \* rightValue;***  ***case '/':***  ***return leftValue / rightValue;***  ***default:***  ***printf("Invalid operator\n");***  ***return 0;***  ***}***  ***}***  ***int main() {***  ***struct Node\* root = createNode('+');***  ***root->left = createNode('5');***  ***root->right = createNode('\*');***  ***root->right->left = createNode('4');***  ***root->right->right = createNode('3');***  ***int result = evaluateExpression(root);***  ***printf("Result of the expression: %d\n", result);***  ***return 0;***  ***}*** |
| **Sample Input:** |
| *Expression: (5 + (4 \* 3))* |
| **Sample Output:** |
| *<* *Result of the expression: 17* |

|  |
| --- |
| **Approach: In-order traversal of tree data structure** |
| **Problem statement** |
| *By performing an in-order traversal, we visit the left subtree first, then the current node, and finally the right subtree. This order ensures that the elements are printed in ascending order as we visit the nodes of the BST* |
| **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 = newNode->right = NULL;***  ***return newNode;***  ***}***  ***struct Node\* insert(struct Node\* root, int data) {***  ***if (root == NULL) {***  ***return createNode(data);***  ***}***  ***if (data < root->data) {***  ***root->left = insert(root->left, data);***  ***} else if (data > root->data) {***  ***root->right = insert(root->right, data);***  ***}***  ***return root;***  ***}***  ***void inOrderTraversal(struct Node\* root) {***  ***if (root == NULL) {***  ***return;***  ***}***  ***inOrderTraversal(root->left);***  ***printf("%d ", root->data);***  ***inOrderTraversal(root->right);***  ***}***  ***int main() {***  ***struct Node\* root = NULL;***  ***root = insert(root, 50);***  ***insert(root, 30);***  ***insert(root, 20);***  *insert(root, 40);*  *insert(root, 70);*  *insert(root, 60);*  *insert(root, 80);*  *printf("In-order traversal of the tree: ");*  *inOrderTraversal(root);*  *printf("\n");*  *return 0;*  *}* |
| **Sample Input:** |
| *50,30,20,40,70,60,80* |
| **Sample Output:** |
| *In-order traversal of the tree: 20 30 40 50 60 70 80* |

|  |
| --- |
| **Approach:** *Post-order traversal of tree data structure* |
| **Problem statement** |
| *Post-order traversal is particularly used memory deallocation in a tree because it ensures that child nodes are deleted before their parent nodes.* |
| **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 = newNode->right = NULL;*  *return newNode;*  *}*  *void postOrderDeletion(struct Node\* root) {*  *if (root == NULL) {*  *return;*  *}*  *postOrderDeletion(root->left);*  *postOrderDeletion(root->right);*  *free(root);*  *}*  *int main() {*  *struct Node\* root = createNode(1);*  *root->left = createNode(2);*  *root->right = createNode(3);*  *root->left->left = createNode(4);*  *root->left->right = createNode(5);*  *printf("Deleting the binary tree using post-order traversal...\n");*  *postOrderDeletion(root);*  *root = NULL; // Reset root pointer after memory deallocation*  *printf("Binary tree has been deleted successfully!\n");*  *return 0;*  *}* |
| **Sample Input:** |
| *1*  */ \*  *2 3*  */ \*  *4 5* |
| **Sample Output:** |
| *Deleting the binary tree using post-order traversal...*  *Binary tree has been deleted successfully!* |

|  |
| --- |
| **Approach: *DFS*** |
| **Problem statement** |
| *Dfs is used for finding connected components in an undirected graph because it efficiently explores connected regions, backtracks to ensure completeness, and marks visited vertices to prevent revisiting* |
| **Code** |
| *<#include <stdio.h>*  *#include <stdlib.h>*  *#include <stdbool.h>*  *#define MAX\_VERTICES 100*  *bool visited[MAX\_VERTICES];*  *struct Graph {*  *int vertices;*  *int\*\* adjacencyMatrix;*  *};*  *struct Graph\* createGraph(int vertices) {*  *struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));*  *graph->vertices = vertices;*  *graph->adjacencyMatrix = (int\*\*)malloc(vertices \* sizeof(int\*));*  *for (int i = 0; i < vertices; i++) {*  *graph->adjacencyMatrix[i] = (int\*)malloc(vertices \* sizeof(int));*  *for (int j = 0; j < vertices; j++) {*  *graph->adjacencyMatrix[i][j] = 0;*  *}*  *}*  *return graph;*  *}*  *void addEdge(struct Graph\* graph, int src, int dest) {*  *graph->adjacencyMatrix[src][dest] = 1;*  *graph->adjacencyMatrix[dest][src] = 1;*  *}*  *void DFS(struct Graph\* graph, int vertex) {*  *visited[vertex] = true;*  *printf("%d ", vertex);*  *for (int i = 0; i < graph->vertices; i++) {*  *if (graph->adjacencyMatrix[vertex][i] == 1 && !visited[i]) {*  *DFS(graph, i);*  *}*  *}*  *}*  *void findConnectedComponents(struct Graph\* graph) {*  *for (int i = 0; i < graph->vertices; i++) {*  *if (!visited[i]) {*  *printf("Connected Component: ");*  *DFS(graph, i);*  *printf("\n");*  *}*  *}*  *}*  *int main() {*  *int numVertices = 7;*  *struct Graph\* graph = createGraph(numVertices);*  *addEdge(graph, 0, 1);*  *addEdge(graph, 0, 2);*  *addEdge(graph, 1, 2);*  *addEdge(graph, 3, 4);*  *addEdge(graph, 5, 6);*  *printf("Connected Components in the graph:\n");*  *findConnectedComponents(graph);*  *return 0;*  *}* |
| **Sample Input:** |
| *0*  */ \*  *1 2*  *|*  *3*  */ \*  *4 5*  */*  *6* |
| **Sample Output:** |
| *Connected Components in the graph:*  *Connected Component: 0 1 2*  *Connected Component: 3 4*  *Connected Component: 5 6* |

|  |
| --- |
| **Approach:** *BFS* |
| **Problem statement** |
| *BFS explores all the vertices at the same level before moving to the next level, which allows it to find the shortest path from the source vertex to any other reachable vertex in an unweighted graph.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_VERTICES 100  struct Queue {  int items[MAX\_VERTICES];  int front;  int rear;  };  struct Queue\* createQueue() {  struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));  queue->front = -1;  queue->rear = -1;  return queue;  }  int isEmpty(struct Queue\* queue) {  return queue->front == -1;  }  void enqueue(struct Queue\* queue, int value) {  if (queue->rear == MAX\_VERTICES - 1) {  printf("Queue is full\n");  } else {  if (queue->front == -1) {  queue->front = 0;  }  queue->rear++;  queue->items[queue->rear] = value;  }  }  int dequeue(struct Queue\* queue) {  int item;  if (isEmpty(queue)) {  printf("Queue is empty\n");  item = -1;  } else {  item = queue->items[queue->front];  queue->front++;  if (queue->front > queue->rear) {  queue->front = queue->rear = -1;  }  }  return item;  }  struct Graph {  int vertices;  int\*\* adjacencyMatrix;  };  struct Graph\* createGraph(int vertices) {  struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));  graph->vertices = vertices;  graph->adjacencyMatrix = (int\*\*)malloc(vertices \* sizeof(int\*));  for (int i = 0; i < vertices; i++) {  graph->adjacencyMatrix[i] = (int\*)malloc(vertices \* sizeof(int));  for (int j = 0; j < vertices; j++) {  graph->adjacencyMatrix[i][j] = 0;  }  }  return graph;  }  void addEdge(struct Graph\* graph, int src, int dest) {  graph->adjacencyMatrix[src][dest] = 1;  graph->adjacencyMatrix[dest][src] = 1;  }  void BFS(struct Graph\* graph, int startVertex, int endVertex) {  int visited[MAX\_VERTICES] = {0};  int parent[MAX\_VERTICES];  for (int i = 0; i < MAX\_VERTICES; i++) {  parent[i] = -1;  }  struct Queue\* queue = createQueue();  visited[startVertex] = 1;  enqueue(queue, startVertex);  while (!isEmpty(queue)) {  int currentVertex = dequeue(queue);  if (currentVertex == endVertex) {  printf("Shortest path found: %d", currentVertex);  int parentVertex = parent[currentVertex];  while (parentVertex != -1) {  printf(" <- %d", parentVertex);  parentVertex = parent[parentVertex];  }  printf("\n");  return;  }  for (int i = 0; i < graph->vertices; i++) {  if (graph->adjacencyMatrix[currentVertex][i] == 1 && !visited[i]) {  visited[i] = 1;  parent[i] = currentVertex;  enqueue(queue, i);  }  }  }  printf("No path found between the given vertices.\n");  }  int main() {  int numVertices = 8;  struct Graph\* graph = createGraph(numVertices);  addEdge(graph, 0, 1);  addEdge(graph, 0, 2);  addEdge(graph, 1, 3);  addEdge(graph, 1, 4);  addEdge(graph, 2, 5);  addEdge(graph, 2, 6);  addEdge(graph, 4, 7);  int startVertex = 0;  int endVertex = 7;  printf("BFS shortest path from vertex %d to vertex %d: \n", startVertex, endVertex);  BFS(graph, startVertex, endVertex);  return 0;  } |
| **Sample Input:** |
| *0*  */ \*  *1 2*  */ \ / \*  *3 4 5 6*  *\*  *7* |
| **Sample Output:** |
| *BFS shortest path from vertex 0 to vertex 7:*  *Shortest path found: 7 <- 4 <- 1 <- 0* |

|  |
| --- |
| **Approach:** *Linear probing* |
| **Problem statement** |
| *When a collision occurs linear probing searches for the next available slot in the hash table by checking the next consecutive position until an empty slot is found.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define TABLE\_SIZE 10  struct HashTableEntry {  int key;  int value;  bool isOccupied;  };  struct HashTable {  struct HashTableEntry table[TABLE\_SIZE];  };  void initializeHashTable(struct HashTable\* hashTable) {  for (int i = 0; i < TABLE\_SIZE; i++) {  hashTable->table[i].key = -1;  hashTable->table[i].isOccupied = false;  }  }  int hashFunction(int key) {  return key % TABLE\_SIZE;  }  void insert(struct HashTable\* hashTable, int key, int value) {  int index = hashFunction(key);  while (hashTable->table[index].isOccupied) {  index = (index + 1) % TABLE\_SIZE;  }  hashTable->table[index].key = key;  hashTable->table[index].value = value;  hashTable->table[index].isOccupied = true;  }  int search(struct HashTable\* hashTable, int key) {  int index = hashFunction(key);  while (hashTable->table[index].isOccupied) {  if (hashTable->table[index].key == key) {  return hashTable->table[index].value;  }  index = (index + 1) % TABLE\_SIZE;  }  return -1;  }  int main() {  struct HashTable hashTable;  initializeHashTable(&hashTable);  insert(&hashTable, 5, 100);  insert(&hashTable, 15, 200);  insert(&hashTable, 25, 300);  insert(&hashTable, 35, 400);  printf("Value at key 15: %d\n", search(&hashTable, 15));  printf("Value at key 25: %d\n", search(&hashTable, 25));  printf("Value at key 10: %d\n", search(&hashTable, 10));  return 0;  } |
| **Sample Input:** |
| *insert(&hashTable, 5, 100);*  *insert(&hashTable, 15, 200);*  *insert(&hashTable, 25, 300);*  *insert(&hashTable, 35, 400);* |
| **Sample Output:** |
| *Value at key 15: 200*  *Value at key 25: 300*  *Value at key 10: -1* |

|  |
| --- |
| **Approach: *Q****uadratic probing* |
| **Problem statement** |
| *Quadratic probing, on the other hand, uses a quadratic function to probe for a new position when a collision occurs. It searches for the next available slot in a quadratic manner which helps to distribute the elements more evenly throughout the hash table. This leads to better utilization of the table and reduces clustering.* |
| **Code** |
| #include <stdio.h>  #include <stdbool.h>  #define SIZE 10 // Size of the hash table  struct HashTable {  int arr[SIZE];  bool occupied[SIZE];  };  int hashFunction(int key) {  return key % SIZE;  }  int quadraticProbe(int key, int i) {  return (hashFunction(key) + i \* i) % SIZE;  }  void insert(struct HashTable\* ht, int key) {  int index = hashFunction(key);  int i = 0;  while (ht->occupied[index]) {  index = quadraticProbe(key, ++i);  }  ht->arr[index] = key;  ht->occupied[index] = true;  }  int main() {  struct HashTable ht = {0};  int keys[] = {19, 6, 14, 23, 2, 9, 20, 30, 11};  for (int i = 0; i < sizeof(keys) / sizeof(keys[0]); i++) {  insert(&ht, keys[i]);  }  printf("Hash Table with Quadratic Probing:\n");  for (int i = 0; i < SIZE; i++) {  if (ht.occupied[i]) {  printf("Index %d: %d\n", i, ht.arr[i]);  } else {  printf("Index %d: Empty\n", i);  }  }  return 0;  } |
| **Sample Input:** |
| *19, 6, 14, 23, 2, 9, 20, 30, 11* |
| **Sample Output:** |
| *Hash Table with Quadratic Probing:*  *Index 0: 0*  *Index 1: 1*  *Index 2: 2*  *Index 3: Empty*  *Index 4: 14*  *Index 5: 19*  *Index 6: 6*  *Index 7: 23*  *Index 8: 30*  *Index 9: Empty* |

|  |
| --- |
| **Approach:** *Double hashing* |
| **Problem statement** |
| *Double hashing does not require additional data structures. It uses the same underlying array for the hash table, which saves memory and simplifies the implementation.* *Double hashing has better space utilization than linear probing* |
| **Code** |
| #include <stdio.h>  #include <stdbool.h>  #define TABLE\_SIZE 10  struct Entry {  int key;  int value;  bool isOccupied;  };  int hash1(int key) {  return key % TABLE\_SIZE;  }  int hash2(int key) {  // A secondary hash function that ensures it never returns 0 to avoid infinite loops  return 1 + (key % (TABLE\_SIZE - 1));  }  int doubleHashingSearch(struct Entry\* hashTable, int key) {  int index = hash1(key);  int step = hash2(key);  int attempts = 0;  while (hashTable[index].isOccupied) {  if (hashTable[index].key == key) {  return index;  }  attempts++;  index = (index + step) % TABLE\_SIZE;  // Prevent infinite loops  if (attempts >= TABLE\_SIZE) {  break;  }  }  return -1; // Key not found  }  bool doubleHashingInsert(struct Entry\* hashTable, int key, int value) {  int index = hash1(key);  int step = hash2(key);  int attempts = 0;  while (hashTable[index].isOccupied) {  if (hashTable[index].key == key) {  return false; // Key already exists  }  attempts++;  index = (index + step) % TABLE\_SIZE;  // Prevent infinite loops  if (attempts >= TABLE\_SIZE) {  return false;  }  }  hashTable[index].key = key;  hashTable[index].value = value;  hashTable[index].isOccupied = true;  return true;  }  int main() {  struct Entry hashTable[TABLE\_SIZE] = {0};  doubleHashingInsert(hashTable, 5, 50);  doubleHashingInsert(hashTable, 15, 150);  doubleHashingInsert(hashTable, 25, 250);  doubleHashingInsert(hashTable, 35, 350);  int keyToSearch = 15;  int searchResult = doubleHashingSearch(hashTable, keyToSearch);  if (searchResult != -1) {  printf("Key %d found at index %d with value %d\n", keyToSearch, searchResult, hashTable[searchResult].value);  } else {  printf("Key %d not found in the hash table.\n", keyToSearch);  }  return 0;  } |
| **Sample Input:** |
| *Insert key 5 with value 50*  *Insert key 15 with value 150*  *Search for key 10*  *Insert key 25 with value 250*  *Search for key 5*  *Insert key 35 with value 350*  *Search for key 15* |
| **Sample Output:** |
| *Inserting key 5 with value 50... Insertion successful.*  *Inserting key 15 with value 150... Insertion successful.*  *Searching for key 10... Key not found in the hash table.*  *Inserting key 25 with value 250... Insertion successful.*  *Searching for key 5... Key found at index 5 with value 50.*  *Inserting key 35 with value 350... Insertion successful.*  *Searching for key 15... Key found at index 5 with value 150.* |