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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *02* | | |  |  | | | |
| **Student Name** | AMOGH | | |  |  | | | |
| **SRN** | 01FE2XBEC122 | | **Roll Number** | | 316 | **Division** | c | |
| **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.construct bst | | | | | | | |
| 2. | | | | | | | |
| **In-order traversal of tree data structure** | 1.kth smallest element in the tree | | | | | | | |
| 2. | | | | | | | |
| **Post-order traversal of tree data structure** | 1.binary expression tree | | | | | | | |
| 2. | | | | | | | |
| **DFS of graphs** | 1.find out the bridges | | | | | | | |
| 2. | | | | | | | |
| **BFS of graphs** | 1.can be coloured using two colours | | | | | | | |
| 2. | | | | | | | |
| **Linear probing of hashing** | 1.insert,search,delete operations | | | | | | | |
| 2. | | | | | | | |
| **Quadratic probing of hashing** | 1.first repeating character | | | | | | | |
| 2. | | | | | | | |
| **Double hashing** | 1.first non repeating string | | | | | | | |
| 2. | | | | | | | |

|  |
| --- |
| **Approach:** *Pre-order traversal of tree data structure* |
| **Problem statement** |
| *Write a C program to construct a BST from an input array of integers using pre-order traversal.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  // Structure for a BST node  struct Node {  int data;  struct Node\* left;  struct Node\* right;  };  // Function to create a new node  struct Node\* newNode(int data) {  struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));  node->data = data;  node->left = NULL;  node->right = NULL;  return node;  }  // Function to construct BST from pre-order traversal  struct Node\* constructBST(int preorder[], int\* index, int min, int max, int size) {  // Base case  if (\*index >= size)  return NULL;  struct Node\* root = NULL;  // If current element of preorder array is within the range of min and max  if (preorder[\*index] > min && preorder[\*index] < max) {  root = newNode(preorder[\*index]);  (\*index)++;  if (\*index < size) {  // Construct left subtree  root->left = constructBST(preorder, index, min, root->data, size);    // Construct right subtree  root->right = constructBST(preorder, index, root->data, max, size);  }  }  return root;  }  // Function to perform in-order traversal of BST  void inorderTraversal(struct Node\* root) {  if (root == NULL)  return;  inorderTraversal(root->left);  printf("%d ", root->data);  inorderTraversal(root->right);  }  int main() {  int preorder[] = { 8, 3, 1, 6, 4, 7, 10, 14, 13 };  int size = sizeof(preorder) / sizeof(preorder[0]);  int index = 0;  struct Node\* root = constructBST(preorder, &index, INT\_MIN, INT\_MAX, size);  printf("In-order traversal of the constructed BST:\n");  inorderTraversal(root);  return 0;  } |
| **Sample Input:** |
| 8, 3, 1, 6, 4, 7, 10, 14, 13 |
| **Sample Output:** |
| *1 3 4 6 7 8 10 13 14* |

|  |
| --- |
| **Approach:** *Pre-order traversal of tree data structure* |
| **Problem statement** |
| *Implement a C function to find the maximum value in a binary tree using pre-order traversal.* |
| **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;  }  int findMaxValue(struct Node\* root) {  int maxValue = root->data;  if (root->left != NULL) {  int leftMax = findMaxValue(root->left);  if (leftMax > maxValue)  maxValue = leftMax;  }  if (root->right != NULL) {  int rightMax = findMaxValue(root->right);  if (rightMax > maxValue)  maxValue = rightMax;  }  return maxValue;  }  int main() {  struct Node\* root = createNode(10);  root->left = createNode(5);  root->right = createNode(15);  root->left->left = createNode(3);  root->left->right = createNode(7);  root->right->left = createNode(12);  root->right->right = createNode(20);  int maxVal = findMaxValue(root);  printf("The maximum value in the binary tree is: %d\n", maxVal);  return 0;  } |
| **Sample Input:** |
| *5,15,3,7,12,20* |
| **Sample Output:** |
| *The maximum value in the binary tree is: 20* |

|  |
| --- |
| **Approach:** *In-order traversal of tree data structure* |
| **Problem statement** |
| *Given a binary search tree, how can you use inorder traversal to find the Kth smallest element in the tree efficiently?* |
| **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;*  *}*  *void findKthSmallest(struct Node\* root, int k, int\* count, int\* result) {*  *if (root == NULL)*  *return;*  *// Traverse the left subtree*  *findKthSmallest(root->left, k, count, result);*  *// Increment the count and check if it matches k*  *(\*count)++;*  *if (\*count == k) {*  *\*result = root->data;*  *return;*  *}*  *// Traverse the right subtree*  *findKthSmallest(root->right, k, count, result);*  *}*  *int findKthSmallestElement(struct Node\* root, int k) {*  *int count = 0;*  *int result = -1;*  *findKthSmallest(root, k, &count, &result);*  *return result;*  *}*  *int main() {*  *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);*  *int k = 3;*  *int kthSmallest = findKthSmallestElement(root, k);*  *if (kthSmallest != -1)*  *printf("The %dth smallest element in the BST is: %d\n", k, kthSmallest);*  *else*  *printf("Invalid value of k\n");*  *return 0;*  *}* |
| **Sample Input:** |
| *5,3,7,2,4,6,8* |
| **Sample Output:** |
| *The 3rd smallest element in the BST is: 4* |

|  |
| --- |
| **Approach:** *In-order traversal of tree data structure* |
| **Problem statement** |
| *Given a binary search tree, how can you use inorder traversal to convert it into a sorted array?* |
| **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;*  *}*  *void inorderTraversal(struct Node\* root, int\* arr, int\* index) {*  *if (root == NULL)*  *return;*  *// Traverse the left subtree*  *inorderTraversal(root->left, arr, index);*  *// Store the current node's data in the array*  *arr[(\*index)++] = root->data;*  *// Traverse the right subtree*  *inorderTraversal(root->right, arr, index);*  *}*  *int\* convertBSTToArray(struct Node\* root, int size) {*  *int\* arr = (int\*)malloc(size \* sizeof(int));*  *int index = 0;*  *inorderTraversal(root, arr, &index);*  *return arr;*  *}*  *int main() {*  *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);*  *int size = 7;*  *int\* sortedArray = convertBSTToArray(root, size);*  *printf("Sorted array representation of the BST:\n");*  *for (int i = 0; i < size; i++) {*  *printf("%d ", sortedArray[i]);*  *}*  *printf("\n");*  *free(sortedArray);*  *return 0;*  *}* |
| **Sample Input:** |
| *5,3,7,2,4,6,8* |
| **Sample Output:** |
| *2 3 4 5 6 7 8* |

|  |
| --- |
| **Approach:** *post-order traversal of tree data structure* |
| **Problem statement** |
| *Write a C program to construct a binary expression tree from a given postfix expression and evaluate it using post-order traversal* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  // Structure for a binary tree node  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 check if a character is an operator  int isOperator(char ch) {  if (ch == '+' || ch == '-' || ch == '\*' || ch == '/')  return 1;  return 0;  }  // Function to construct a binary expression tree from a postfix expression  struct Node\* constructExpressionTree(char postfix[]) {  struct Node\* stack[100];  int top = -1;  for (int i = 0; postfix[i] != '\0'; i++) {  if (isOperator(postfix[i])) {  struct Node\* node = createNode(postfix[i]);  node->right = stack[top--];  node->left = stack[top--];  stack[++top] = node;  } else {  struct Node\* node = createNode(postfix[i]);  stack[++top] = node;  }  }  return stack[top--];  }  // Function to evaluate the binary expression tree using post-order traversal  int evaluateExpressionTree(struct Node\* root) {  if (root->left == NULL && root->right == NULL)  return root->data - '0';  int leftVal = evaluateExpressionTree(root->left);  int rightVal = evaluateExpressionTree(root->right);  switch (root->data) {  case '+':  return leftVal + rightVal;  case '-':  return leftVal - rightVal;  case '\*':  return leftVal \* rightVal;  case '/':  return leftVal / rightVal;  }  return 0;  }  int main() {  char postfix[] = "34+52-\*";  struct Node\* root = constructExpressionTree(postfix);  int result = evaluateExpressionTree(root);  printf("Result: %d\n", result);  return 0;  } |
| **Sample Input:** |
| *34+52-\** |
| **Sample Output:** |
| *Result: 9* |

|  |
| --- |
| **Approach:** *post-order traversal of tree data structure* |
| **Problem statement** |
| *Given a binary tree, write a C program to find the maximum element in the tree using post-order traversal.* |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  // Structure for a binary tree node  struct Node {  int data;  struct Node\* left;  struct Node\* right;  };  // Function to create a new binary tree 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 post-order traversal and find the maximum element  int findMaxElement(struct Node\* root) {  if (root == NULL) {  return INT\_MIN; // Assuming tree nodes contain positive integers  }  int leftMax = findMaxElement(root->left);  int rightMax = findMaxElement(root->right);  int max = root->data;  if (leftMax > max) {  max = leftMax;  }  if (rightMax > max) {  max = rightMax;  }  return max;  }  int main() {  // Create the binary tree  struct Node\* root = createNode(2);  root->left = createNode(7);  root->right = createNode(5);  root->left->left = createNode(2);  root->left->right = createNode(6);  root->left->right->left = createNode(5);  root->left->right->right = createNode(11);  root->right->right = createNode(9);  root->right->right->left = createNode(4);  // Find the maximum element  int maxElement = findMaxElement(root);  printf("The maximum element in the tree is: %d\n", maxElement);  return 0;  } |
| **Sample Input:** |
| *2,7,5,2,6,5,11,9,4* |
| **Sample Output:** |
| *The maximum element in the tree is: 11* |

|  |
| --- |
| **Approach:** *DFS* |
| **Problem statement** |
| *Given an undirected graph, write a C program to find the bridges (cut edges) using DFS* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#define MAX\_VERTICES 100*  *int time = 0;*  *// Structure to represent an edge*  *struct Edge {*  *int src, dest;*  *};*  *// Structure to represent a graph*  *struct Graph {*  *int numVertices, numEdges;*  *struct Edge\* edges;*  *};*  *// Function to perform DFS and find bridges*  *void findBridges(int u, int parent, int disc[], int low[], int visited[], struct Graph\* graph) {*  *// Mark the current vertex as visited*  *visited[u] = 1;*  *// Initialize discovery time and low value*  *disc[u] = low[u] = ++time;*  *// Iterate through all adjacent vertices of u*  *for (int i = 0; i < graph->numEdges; i++) {*  *int v = graph->edges[i].dest;*    *// If v is not visited, recursively visit it*  *if (graph->edges[i].src == u && visited[v] == 0) {*  *findBridges(v, u, disc, low, visited, graph);*  *// Check if the subtree rooted at v has a connection to ancestors of u*  *low[u] = (low[u] < low[v]) ? low[u] : low[v];*  *// If the lowest vertex reachable from the subtree rooted at v*  *// is below u in the DFS tree, then (u, v) is a bridge*  *if (low[v] > disc[u])*  *printf("(%d, %d) is a bridge\n", u, v);*  *}*  *// Update low value of u for the case of a backward edge*  *else if (v != parent)*  *low[u] = (low[u] < disc[v]) ? low[u] : disc[v];*  *}*  *}*  *// Function to find bridges in an undirected graph using DFS*  *void findBridgesDFS(struct Graph\* graph) {*  *int\* disc = (int\*)malloc(graph->numVertices \* sizeof(int));*  *int\* low = (int\*)malloc(graph->numVertices \* sizeof(int));*  *int\* visited = (int\*)malloc(graph->numVertices \* sizeof(int));*  *// Initialize visited array*  *for (int i = 0; i < graph->numVertices; i++)*  *visited[i] = 0;*  *// Perform DFS traversal and find bridges*  *for (int i = 0; i < graph->numVertices; i++) {*  *if (visited[i] == 0)*  *findBridges(i, -1, disc, low, visited, graph);*  *}*  *free(disc);*  *free(low);*  *free(visited);*  *}*  *int main() {*  *struct Graph graph;*  *graph.numVertices = 4;*  *graph.numEdges = 3;*  *graph.edges = (struct Edge\*)malloc(graph.numEdges \* sizeof(struct Edge));*    *// Edge 0-1*  *graph.edges[0].src = 0;*  *graph.edges[0].dest = 1;*  *// Edge 1-2*  *graph.edges[1].src = 1;*  *graph.edges[1].dest = 2;*  *// Edge 2-3*  *graph.edges[2].src = 2;*  *graph.edges[2].dest = 3;*  *findBridgesDFS(&graph);*  *free(graph.edges);*  *return 0;*  *}* |
| **Sample Input:** |
|  |
| **Sample Output:** |
| *(2, 3) is a bridge* |

|  |
| --- |
| **Approach:** *DFS* |
| **Problem statement** |
| *Given a weighted directed graph, write a C program to find the minimum spanning tree (MST) using DFS* |
| **Code** |
| *<#include <stdio.h>*  *#define V 5 // Number of vertices in the graph*  *// Function to find the vertex with the minimum key value*  *int findMinKey(int key[], int visited[]) {*  *int min = 9999, minIndex = -1;*  *for (int v = 0; v < V; v++) {*  *if (!visited[v] && key[v] < min) {*  *min = key[v];*  *minIndex = v;*  *}*  *}*  *return minIndex;*  *}*  *// Function to print the Minimum Spanning Tree*  *void printMST(int parent[], int graph[V][V]) {*  *printf("Edge \tWeight\n");*  *for (int i = 1; i < V; i++) {*  *printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);*  *}*  *}*  *// Function to find the Minimum Spanning Tree using DFS-like approach*  *void findMST(int graph[V][V]) {*  *int parent[V];*  *int key[V];*  *int visited[V];*  *// Initialize key values and visited array*  *for (int i = 0; i < V; i++) {*  *key[i] = 9999;*  *visited[i] = 0;*  *}*  *// Start with the first vertex*  *parent[0] = -1;*  *key[0] = 0;*  *for (int count = 0; count < V - 1; count++) {*  *int u = findMinKey(key, visited);*  *visited[u] = 1;*  *for (int v = 0; v < V; v++) {*  *if (graph[u][v] && !visited[v] && graph[u][v] < key[v]) {*  *parent[v] = u;*  *key[v] = graph[u][v];*  *}*  *}*  *}*  *// Print the Minimum Spanning Tree*  *printMST(parent, graph);*  *}*  *int main() {*  *int graph[V][V] = {*  *{0, 2, 0, 6, 0},*  *{2, 0, 3, 8, 5},*  *{0, 3, 0, 0, 7},*  *{6, 8, 0, 0, 9},*  *{0, 5, 7, 9, 0},*  *};*  *findMST(graph);*  *return 0;*  *}* |
| **Sample Input:** |
|  |
| **Sample Output:** |
| *Edge Weight*  *0 - 1 2*  *1 - 2 3*  *0 - 3 6*  *1 - 4 5* |

|  |
| --- |
| **Approach:** *BFS* |
| **Problem statement** |
| *Given an undirected graph, write a C program to check if it is bipartite (can be colored using two colors) using BFS* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#define MAX\_VERTICES 100*  *// Structure to represent a node in the graph*  *struct Node {*  *int data;*  *struct Node\* next;*  *};*  *// Structure to represent the graph*  *struct Graph {*  *int numVertices;*  *struct Node\*\* adjLists;*  *};*  *// Function to create a new node*  *struct Node\* createNode(int data) {*  *struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));*  *newNode->data = data;*  *newNode->next = NULL;*  *return newNode;*  *}*  *// Function to add an edge to the 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->adjLists[src];*  *graph->adjLists[src] = newNode;*  *// Add an edge from dest to src*  *newNode = createNode(src);*  *newNode->next = graph->adjLists[dest];*  *graph->adjLists[dest] = newNode;*  *}*  *// Function to check if the graph is bipartite using BFS*  *int isBipartite(struct Graph\* graph, int startVertex) {*  *int colorArr[MAX\_VERTICES]; // Array to store the color of each vertex*  *for (int i = 0; i < graph->numVertices; i++) {*  *colorArr[i] = -1; // Initialize the color array with -1 (not colored)*  *}*  *colorArr[startVertex] = 1; // Color the starting vertex as 1*  *// Create a queue for BFS*  *int queue[MAX\_VERTICES];*  *int front = -1, rear = -1;*  *queue[++rear] = startVertex;*  *while (front != rear) {*  *int currentVertex = queue[++front];*  *struct Node\* temp = graph->adjLists[currentVertex];*  *while (temp) {*  *int adjVertex = temp->data;*  *// If the adjacent vertex is not colored, color it with a different color*  *if (colorArr[adjVertex] == -1) {*  *colorArr[adjVertex] = 1 - colorArr[currentVertex];*  *queue[++rear] = adjVertex;*  *}*  *// If the adjacent vertex is already colored with the same color as the current vertex,*  *// then the graph is not bipartite*  *else if (colorArr[adjVertex] == colorArr[currentVertex]) {*  *return 0;*  *}*  *temp = temp->next;*  *}*  *}*  *return 1; // If all vertices can be colored without conflict, the graph is bipartite*  *}*  *int main() {*  *struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));*  *graph->numVertices = 6;*  *graph->adjLists = (struct Node\*)malloc(graph->numVertices \* sizeof(struct Node));*  *for (int i = 0; i < graph->numVertices; i++) {*  *graph->adjLists[i] = NULL;*  *}*  *// Add edges to the graph*  *addEdge(graph, 0, 1);*  *addEdge(graph, 0, 2);*  *addEdge(graph, 1, 3);*  *addEdge(graph, 2, 4);*  *addEdge(graph, 3, 5);*  *addEdge(graph, 4, 5);*  *int startVertex = 0; // Starting vertex for BFS*  *if (isBipartite(graph, startVertex)) {*  *printf("The graph is bipartite.\n");*  *} else {*  *printf("The graph is not bipartite.\n");*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *addEdge(graph, 0, 1);*  *addEdge(graph, 0, 2);*  *addEdge(graph, 1, 3);*  *addEdge(graph, 2, 4);*  *addEdge(graph, 3, 5);*  *addEdge(graph, 4, 5);* |
| **Sample Output:** |
| *The graph is bipartite.* |

|  |
| --- |
| **Approach:** *BFS* |
| **Problem statement** |
| *Given an undirected graph, write a C program to find the centroid of the graph (the node that minimizes the maximum distance to other nodes) using BFS* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#define MAX\_VERTICES 100*  *// Structure to represent a node in the graph*  *struct Node {*  *int data;*  *struct Node\* next;*  *};*  *// Structure to represent the graph*  *struct Graph {*  *int numVertices;*  *struct Node\*\* adjLists;*  *};*  *// Function to create a new node*  *struct Node\* createNode(int data) {*  *struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));*  *newNode->data = data;*  *newNode->next = NULL;*  *return newNode;*  *}*  *// Function to add an edge to the 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->adjLists[src];*  *graph->adjLists[src] = newNode;*  *// Add an edge from dest to src*  *newNode = createNode(src);*  *newNode->next = graph->adjLists[dest];*  *graph->adjLists[dest] = newNode;*  *}*  *// Function to find the centroid of the tree using BFS*  *int findCentroid(struct Graph\* graph, int startVertex) {*  *int numVertices = graph->numVertices;*  *int\* subtreeSize = (int\*)malloc(numVertices \* sizeof(int));*  *int\* visited = (int\*)malloc(numVertices \* sizeof(int));*  *int\* maxSubtreeSize = (int\*)malloc(numVertices \* sizeof(int));*  *int\* parent = (int\*)malloc(numVertices \* sizeof(int));*  *int queue[MAX\_VERTICES];*  *int front = -1, rear = -1;*  *int i, currentVertex, child, maxSubtree, centroid;*  *// Initialize arrays*  *for (i = 0; i < numVertices; i++) {*  *subtreeSize[i] = 1;*  *visited[i] = 0;*  *maxSubtreeSize[i] = 0;*  *parent[i] = -1;*  *}*  *// Perform BFS to calculate subtree sizes*  *queue[++rear] = startVertex;*  *visited[startVertex] = 1;*  *while (front != rear) {*  *currentVertex = queue[++front];*  *struct Node\* temp = graph->adjLists[currentVertex];*  *while (temp) {*  *int adjVertex = temp->data;*  *if (!visited[adjVertex]) {*  *visited[adjVertex] = 1;*  *parent[adjVertex] = currentVertex;*  *queue[++rear] = adjVertex;*  *}*  *temp = temp->next;*  *}*  *}*  *// Calculate subtree sizes*  *for (i = numVertices - 1; i >= 0; i--) {*  *currentVertex = queue[i];*  *struct Node\* temp = graph->adjLists[currentVertex];*  *while (temp) {*  *int adjVertex = temp->data;*  *if (adjVertex != parent[currentVertex]) {*  *subtreeSize[currentVertex] += subtreeSize[adjVertex];*  *maxSubtreeSize[currentVertex] = (maxSubtreeSize[currentVertex] > subtreeSize[adjVertex]) ? maxSubtreeSize[currentVertex] : subtreeSize[adjVertex];*  *}*  *temp = temp->next;*  *}*  *maxSubtreeSize[currentVertex] = (maxSubtreeSize[currentVertex] > numVertices - subtreeSize[currentVertex]) ? maxSubtreeSize[currentVertex] : numVertices - subtreeSize[currentVertex];*  *}*  *// Find the centroid*  *centroid = startVertex;*  *maxSubtree = maxSubtreeSize[startVertex];*  *for (i = 0; i < numVertices; i++) {*  *if (maxSubtreeSize[i] < maxSubtree) {*  *maxSubtree = maxSubtreeSize[i];*  *centroid = i;*  *}*  *}*  *free(subtreeSize);*  *free(visited);*  *free(maxSubtreeSize);*  *free(parent);*  *return centroid;*  *}*  *int main() {*  *struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));*  *graph->numVertices = 6;*  *graph->adjLists = (struct Node\*)malloc(graph->numVertices \* sizeof(struct Node));*  *for (int i = 0; i < graph->numVertices; i++) {*  *graph->adjLists[i] = NULL;*  *}*  *// Add edges to the graph*  *addEdge(graph, 0, 1);*  *addEdge(graph, 0, 2);*  *addEdge(graph, 0, 3);*  *addEdge(graph, 2, 4);*  *addEdge(graph, 3, 5);*  *int centroid = findCentroid(graph, 0);*  *printf("The centroid of the tree is: %d\n", centroid);*  *return 0;*  *}* |
| **Sample Input:** |
| *addEdge(graph, 0, 1);*  *addEdge(graph, 0, 2);*  *addEdge(graph, 0, 3);*  *addEdge(graph, 2, 4);*  *addEdge(graph, 3, 5);* |
| **Sample Output:** |
| *The centroid of the tree is: 0* |

|  |
| --- |
| **Approach:** *Linear probing* |
| **Problem statement** |
| *Write a C program to implement a hash table using linear probing and perform insert, search, and delete operations* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#define SIZE 10 // Size of the hash table*  *// Structure to represent a key-value pair*  *struct KeyValue {*  *int key;*  *int value;*  *int status; // 0: Empty, 1: Occupied, 2: Deleted*  *};*  *// Structure to represent the hash table*  *struct HashTable {*  *struct KeyValue\* array;*  *int capacity;*  *};*  *// Function to create a new hash table*  *struct HashTable\* createHashTable() {*  *struct HashTable\* hashtable = (struct HashTable\*)malloc(sizeof(struct HashTable));*  *hashtable->array = (struct KeyValue\*)malloc(SIZE \* sizeof(struct KeyValue));*  *hashtable->capacity = SIZE;*  *// Initialize all entries as empty*  *for (int i = 0; i < SIZE; i++) {*  *hashtable->array[i].key = -1;*  *hashtable->array[i].status = 0;*  *}*  *return hashtable;*  *}*  *// Hash function to generate an index*  *int hashFunction(int key) {*  *return key % SIZE;*  *}*  *// Function to insert a key-value pair into the hash table*  *void insert(struct HashTable\* hashtable, int key, int value) {*  *int index = hashFunction(key);*  *int i = index;*  *// Find the next empty or deleted slot using linear probing*  *while (hashtable->array[i].status == 1) {*  *if (hashtable->array[i].key == key) {*  *// Key already exists, update the value*  *hashtable->array[i].value = value;*  *return;*  *}*  *i = (i + 1) % SIZE;*  *if (i == index) {*  *// Hash table is full*  *printf("Hash table is full. Cannot insert key-value pair.\n");*  *return;*  *}*  *}*  *// Insert the new key-value pair*  *hashtable->array[i].key = key;*  *hashtable->array[i].value = value;*  *hashtable->array[i].status = 1; // Mark the slot as occupied*  *}*  *// Function to search for a key in the hash table*  *int search(struct HashTable\* hashtable, int key) {*  *int index = hashFunction(key);*  *int i = index;*  *// Find the slot containing the key using linear probing*  *while (hashtable->array[i].status != 0) {*  *if (hashtable->array[i].key == key && hashtable->array[i].status == 1) {*  *return hashtable->array[i].value; // Key found, return the corresponding value*  *}*  *i = (i + 1) % SIZE;*  *if (i == index) {*  *// Reached the original slot, key not found*  *return -1;*  *}*  *}*  *return -1; // Key not found*  *}*  *// Function to delete a key-value pair from the hash table*  *void delete(struct HashTable\* hashtable, int key) {*  *int index = hashFunction(key);*  *int i = index;*  *// Find the slot containing the key using linear probing*  *while (hashtable->array[i].status != 0) {*  *if (hashtable->array[i].key == key && hashtable->array[i].status == 1) {*  *hashtable->array[i].key = -1; // Mark the slot as deleted*  *hashtable->array[i].status = 2; // Mark the slot as deleted*  *return;*  *}*  *i = (i + 1) % SIZE;*  *if (i == index) {*  *// Reached the original slot, key not found*  *return;*  *}*  *}*  *}*  *int main() {*  *struct HashTable\* hashtable = createHashTable();*  *insert(hashtable, 10, 100);*  *insert(hashtable, 20, 200);*  *insert(hashtable, 30, 300);*  *int value = search(hashtable, 20);*  *if (value != -1) {*  *printf("Value found: %d\n", value);*  *} else {*  *printf("Value not found.\n");*  *}*  *delete(hashtable, 20);*  *value = search(hashtable, 20);*  *if (value != -1) {*  *printf("Value found: %d\n", value);*  *} else {*  *printf("Value not found.\n");*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *insert(hashtable, 10, 100);*  *insert(hashtable, 20, 200);*  *insert(hashtable, 30, 300);* |
| **Sample Output:** |
| *Value found: 200*  *Value not found.* |

|  |
| --- |
| **Approach:** *Linear probing* |
| **Problem statement** |
| *Write a C program to find the most frequent element in an array using a hash table with linear probing* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#define SIZE 100 // Size of the hash table*  *// Structure to represent a key-value pair*  *struct KeyValue {*  *int key;*  *int value;*  *int status; // 0: Empty, 1: Occupied, 2: Deleted*  *};*  *// Structure to represent the hash table*  *struct HashTable {*  *struct KeyValue\* array;*  *int capacity;*  *};*  *// Function to create a new hash table*  *struct HashTable\* createHashTable() {*  *struct HashTable\* hashtable = (struct HashTable\*)malloc(sizeof(struct HashTable));*  *hashtable->array = (struct KeyValue\*)malloc(SIZE \* sizeof(struct KeyValue));*  *hashtable->capacity = SIZE;*  *// Initialize all entries as empty*  *for (int i = 0; i < SIZE; i++) {*  *hashtable->array[i].key = -1;*  *hashtable->array[i].status = 0;*  *}*  *return hashtable;*  *}*  *// Hash function to generate an index*  *int hashFunction(int key) {*  *return key % SIZE;*  *}*  *// Function to insert a key-value pair into the hash table*  *void insert(struct HashTable\* hashtable, int key, int value) {*  *int index = hashFunction(key);*  *int i = index;*  *// Find the next empty or deleted slot using linear probing*  *while (hashtable->array[i].status == 1) {*  *if (hashtable->array[i].key == key) {*  *// Key already exists, update the value*  *hashtable->array[i].value = value;*  *return;*  *}*  *i = (i + 1) % SIZE;*  *if (i == index) {*  *// Hash table is full*  *printf("Hash table is full. Cannot insert key-value pair.\n");*  *return;*  *}*  *}*  *// Insert the new key-value pair*  *hashtable->array[i].key = key;*  *hashtable->array[i].value = value;*  *hashtable->array[i].status = 1; // Mark the slot as occupied*  *}*  *// Function to find the most frequent element in the array*  *int findMostFrequentElement(int arr[], int n) {*  *struct HashTable\* hashtable = createHashTable();*  *// Insert each element from the array into the hash table*  *for (int i = 0; i < n; i++) {*  *int key = arr[i];*  *int value = 1;*  *// If the element already exists in the hash table, increment its frequency*  *if (hashtable->array[key].status == 1) {*  *value += hashtable->array[key].value;*  *}*  *insert(hashtable, key, value);*  *}*  *int mostFrequentElement = -1;*  *int maxFrequency = 0;*  *// Find the element with the maximum frequency in the hash table*  *for (int i = 0; i < SIZE; i++) {*  *if (hashtable->array[i].status == 1 && hashtable->array[i].value > maxFrequency) {*  *maxFrequency = hashtable->array[i].value;*  *mostFrequentElement = hashtable->array[i].key;*  *}*  *}*  *free(hashtable->array);*  *free(hashtable);*  *return mostFrequentElement;*  *}*  *int main() {*  *int arr[] = {1, 3, 2, 4, 1, 3, 2, 1, 3, 1, 5};*  *int n = sizeof(arr) / sizeof(arr[0]);*  *int mostFrequentElement = findMostFrequentElement(arr, n);*  *if (mostFrequentElement != -1) {*  *printf("The most frequent element is: %d\n", mostFrequentElement);*  *} else {*  *printf("Array is empty.\n");*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *1, 3, 2, 4, 1, 3, 2, 1, 3, 1, 5* |
| **Sample Output:** |
| *The most frequent element is: 1* |

|  |
| --- |
| **Approach:** *quadratic probing* |
| **Problem statement** |
| *Given a string, write a C program to find the first repeating character using a hash table with quadratic probing* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#define SIZE 256 // Size of the hash table*  *// Structure to represent a key-value pair*  *struct KeyValue {*  *char key;*  *int value;*  *int status; // 0: Empty, 1: Occupied, 2: Deleted*  *};*  *// Structure to represent the hash table*  *struct HashTable {*  *struct KeyValue\* array;*  *int capacity;*  *};*  *// Function to create a new hash table*  *struct HashTable\* createHashTable() {*  *struct HashTable\* hashtable = (struct HashTable\*)malloc(sizeof(struct HashTable));*  *hashtable->array = (struct KeyValue\*)malloc(SIZE \* sizeof(struct KeyValue));*  *hashtable->capacity = SIZE;*  *// Initialize all entries as empty*  *for (int i = 0; i < SIZE; i++) {*  *hashtable->array[i].key = '\0';*  *hashtable->array[i].status = 0;*  *}*  *return hashtable;*  *}*  *// Hash function to generate an index*  *int hashFunction(char key) {*  *return (int)key % SIZE;*  *}*  *// Function to insert a key-value pair into the hash table*  *void insert(struct HashTable\* hashtable, char key, int value) {*  *int index = hashFunction(key);*  *int i = index;*  *// Find the next empty or deleted slot using quadratic probing*  *int step = 1;*  *while (hashtable->array[i].status == 1) {*  *if (hashtable->array[i].key == key) {*  *// Key already exists, update the value*  *hashtable->array[i].value = value;*  *return;*  *}*  *i = (index + step \* step) % SIZE;*  *step++;*  *if (step > SIZE) {*  *// Hash table is full*  *printf("Hash table is full. Cannot insert key-value pair.\n");*  *return;*  *}*  *}*  *// Insert the new key-value pair*  *hashtable->array[i].key = key;*  *hashtable->array[i].value = value;*  *hashtable->array[i].status = 1; // Mark the slot as occupied*  *}*  *// Function to find the first repeating character in a string*  *char findFirstRepeatingChar(char\* str) {*  *struct HashTable\* hashtable = createHashTable();*  *// Traverse the string and insert each character into the hash table*  *for (int i = 0; str[i] != '\0'; i++) {*  *char key = str[i];*  *int value = i;*  *// If the character already exists in the hash table, return it as the first repeating character*  *if (hashtable->array[key].status == 1) {*  *free(hashtable->array);*  *free(hashtable);*  *return key;*  *}*  *insert(hashtable, key, value);*  *}*  *free(hashtable->array);*  *free(hashtable);*  *return '\0'; // No repeating character found*  *}*  *int main() {*  *char str[] = "Hello, World!";*  *char firstRepeatingChar = findFirstRepeatingChar(str);*  *if (firstRepeatingChar != '\0') {*  *printf("The first repeating character is: %c\n", firstRepeatingChar);*  *} else {*  *printf("No repeating character found.\n");*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *"Hello, World!"* |
| **Sample Output:** |
| *The first repeating character is: l* |

|  |
| --- |
| **Approach:** *quadratic probing* |
| **Problem statement** |
| *Given an array of integers, write a C program to find the subarray with the maximum sum using a hash table with quadratic probing.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#define SIZE 100 // Size of the hash table*  *// Structure to represent a key-value pair*  *struct KeyValue {*  *int key;*  *int value;*  *int status; // 0: Empty, 1: Occupied, 2: Deleted*  *};*  *// Structure to represent the hash table*  *struct HashTable {*  *struct KeyValue\* array;*  *int capacity;*  *};*  *// Function to create a new hash table*  *struct HashTable\* createHashTable() {*  *struct HashTable\* hashtable = (struct HashTable\*)malloc(sizeof(struct HashTable));*  *hashtable->array = (struct KeyValue\*)malloc(SIZE \* sizeof(struct KeyValue));*  *hashtable->capacity = SIZE;*  *// Initialize all entries as empty*  *for (int i = 0; i < SIZE; i++) {*  *hashtable->array[i].key = -1;*  *hashtable->array[i].status = 0;*  *}*  *return hashtable;*  *}*  *// Hash function to generate an index*  *int hashFunction(int key) {*  *return key % SIZE;*  *}*  *// Function to insert a key-value pair into the hash table*  *void insert(struct HashTable\* hashtable, int key, int value) {*  *int index = hashFunction(key);*  *int i = index;*  *// Find the next empty or deleted slot using quadratic probing*  *int step = 1;*  *while (hashtable->array[i].status == 1) {*  *if (hashtable->array[i].key == key) {*  *// Key already exists, update the value*  *hashtable->array[i].value = value;*  *return;*  *}*  *i = (index + step \* step) % SIZE;*  *step++;*  *if (step > SIZE) {*  *// Hash table is full*  *printf("Hash table is full. Cannot insert key-value pair.\n");*  *return;*  *}*  *}*  *// Insert the new key-value pair*  *hashtable->array[i].key = key;*  *hashtable->array[i].value = value;*  *hashtable->array[i].status = 1; // Mark the slot as occupied*  *}*  *// Function to find the subarray with the maximum sum in an array*  *void findMaxSumSubarray(int arr[], int n) {*  *struct HashTable\* hashtable = createHashTable();*  *int maxSum = 0;*  *int startIndex = -1;*  *int endIndex = -1;*  *int currentSum = 0;*  *// Traverse the array and find the subarray with the maximum sum*  *for (int i = 0; i < n; i++) {*  *currentSum += arr[i];*  *// If currentSum becomes negative, reset it to 0*  *if (currentSum < 0) {*  *currentSum = 0;*  *startIndex = i + 1; // Update the startIndex*  *}*  *// If currentSum is greater than maxSum, update maxSum and endIndex*  *if (currentSum > maxSum) {*  *maxSum = currentSum;*  *endIndex = i;*  *}*  *}*  *// If all elements are negative, find the maximum element as the subarray*  *if (endIndex == -1) {*  *int maxElement = arr[0];*  *for (int i = 1; i < n; i++) {*  *if (arr[i] > maxElement) {*  *maxElement = arr[i];*  *startIndex = i;*  *endIndex = i;*  *}*  *}*  *}*  *printf("Subarray with the maximum sum: ");*  *for (int i = startIndex; i <= endIndex; i++) {*  *printf("%d ", arr[i]);*  *}*  *printf("\n");*  *free(hashtable->array);*  *free(hashtable);*  *}*  *int main() {*  *int arr[] = {-2, -3, 4, -1, -2, 1, 5, -3};*  *int n = sizeof(arr) / sizeof(arr[0]);*  *findMaxSumSubarray(arr, n);*  *return 0;*  *}* |
| **Sample Input:** |
| *-2, -3, 4, -1, -2, 1, 5, -3* |
| **Sample Output:** |
| *Subarray with the maximum sum: 4 -1 -2 1 5* |

|  |
| --- |
| **Approach:** *double hashing* |
| **Problem statement** |
| *Given an array of strings, write a C program to find the first non-repeating string using a hash table with double hashing.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <string.h>*  *#define SIZE 100 // Size of the hash table*  *// Structure to represent a key-value pair*  *struct KeyValue {*  *char\* key;*  *int value;*  *int status; // 0: Empty, 1: Occupied, 2: Deleted*  *};*  *// Structure to represent the hash table*  *struct HashTable {*  *struct KeyValue\* array;*  *int capacity;*  *};*  *// Function to create a new hash table*  *struct HashTable\* createHashTable() {*  *struct HashTable\* hashtable = (struct HashTable\*)malloc(sizeof(struct HashTable));*  *hashtable->array = (struct KeyValue\*)malloc(SIZE \* sizeof(struct KeyValue));*  *hashtable->capacity = SIZE;*  *// Initialize all entries as empty*  *for (int i = 0; i < SIZE; i++) {*  *hashtable->array[i].key = NULL;*  *hashtable->array[i].status = 0;*  *}*  *return hashtable;*  *}*  *// Hash function to generate an index*  *int hashFunction1(char\* key) {*  *int hash = 0;*  *for (int i = 0; key[i] != '\0'; i++) {*  *hash = (hash \* 31 + key[i]) % SIZE;*  *}*  *return hash;*  *}*  *// Hash function for double hashing*  *int hashFunction2(char\* key) {*  *int hash = 0;*  *for (int i = 0; key[i] != '\0'; i++) {*  *hash += key[i];*  *}*  *return hash % (SIZE - 1) + 1;*  *}*  *// Function to insert a key-value pair into the hash table*  *void insert(struct HashTable\* hashtable, char\* key, int value) {*  *int index1 = hashFunction1(key);*  *int index2 = hashFunction2(key);*  *int index = index1;*  *// Find the next empty or deleted slot using double hashing*  *int step = 0;*  *while (hashtable->array[index].status == 1) {*  *if (strcmp(hashtable->array[index].key, key) == 0) {*  *// Key already exists, update the value*  *hashtable->array[index].value = value;*  *return;*  *}*  *step++;*  *index = (index1 + step \* index2) % SIZE;*  *if (step > SIZE) {*  *// Hash table is full*  *printf("Hash table is full. Cannot insert key-value pair.\n");*  *return;*  *}*  *}*  *// Insert the new key-value pair*  *hashtable->array[index].key = key;*  *hashtable->array[index].value = value;*  *hashtable->array[index].status = 1; // Mark the slot as occupied*  *}*  *// Function to find the first non-repeating string in the array*  *char\* findFirstNonRepeatingString(char\* arr[], int n) {*  *struct HashTable\* hashtable = createHashTable();*  *// Traverse the array and insert each string into the hash table*  *for (int i = 0; i < n; i++) {*  *char\* key = arr[i];*  *int value = i;*  *// If the string already exists in the hash table, mark it as repeating*  *if (hashtable->array[hashFunction1(key)].status == 1 &&*  *strcmp(hashtable->array[hashFunction1(key)].key, key) == 0) {*  *hashtable->array[hashFunction1(key)].status = 2;*  *} else {*  *insert(hashtable, key, value);*  *}*  *}*  *// Find the first non-repeating string in the hash table*  *for (int i = 0; i < SIZE; i++) {*  *if (hashtable->array[i].status == 1) {*  *free(hashtable->array);*  *free(hashtable);*  *return hashtable->array[i].key;*  *}*  *}*  *free(hashtable->array);*  *free(hashtable);*  *return NULL; // No non-repeating string found*  *}*  *int main() {*  *char\* arr[] = {"apple", "banana", "orange", "apple", "banana", "kiwi"};*  *int n = sizeof(arr) / sizeof(arr[0]);*  *char\* firstNonRepeatingString = findFirstNonRepeatingString(arr, n);*  *if (firstNonRepeatingString != NULL) {*  *printf("The first non-repeating string is: %s\n", firstNonRepeatingString);*  *} else {*  *printf("No non-repeating string found.\n");*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *“apple", "banana", "orange", "apple", "banana", "kiwi"* |
| **Sample Output:** |
| *The first non-repeating string is: orange* |

|  |
| --- |
| **Approach:** *double hashing* |
| **Problem statement** |
| *Write a C program to check if an array contains duplicate elements using a hash table with double hashing.* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#define SIZE 100 // Size of the hash table*  *// Structure to represent a key-value pair*  *struct KeyValue {*  *int key;*  *int value;*  *int status; // 0: Empty, 1: Occupied, 2: Deleted*  *};*  *// Structure to represent the hash table*  *struct HashTable {*  *struct KeyValue\* array;*  *int capacity;*  *};*  *// Function to create a new hash table*  *struct HashTable\* createHashTable() {*  *struct HashTable\* hashtable = (struct HashTable\*)malloc(sizeof(struct HashTable));*  *hashtable->array = (struct KeyValue\*)malloc(SIZE \* sizeof(struct KeyValue));*  *hashtable->capacity = SIZE;*  *// Initialize all entries as empty*  *for (int i = 0; i < SIZE; i++) {*  *hashtable->array[i].key = -1;*  *hashtable->array[i].status = 0;*  *}*  *return hashtable;*  *}*  *// Hash function to generate an index*  *int hashFunction1(int key) {*  *return key % SIZE;*  *}*  *// Hash function for double hashing*  *int hashFunction2(int key) {*  *return 1 + (key % (SIZE - 1));*  *}*  *// Function to insert a key-value pair into the hash table*  *void insert(struct HashTable\* hashtable, int key, int value) {*  *int index1 = hashFunction1(key);*  *int index2 = hashFunction2(key);*  *int index = index1;*  *// Find the next empty or deleted slot using double hashing*  *int step = 0;*  *while (hashtable->array[index].status == 1) {*  *if (hashtable->array[index].key == key) {*  *// Key already exists, update the value*  *hashtable->array[index].value = value;*  *return;*  *}*  *step++;*  *index = (index1 + step \* index2) % SIZE;*  *if (step > SIZE) {*  *// Hash table is full*  *printf("Hash table is full. Cannot insert key-value pair.\n");*  *return;*  *}*  *}*  *// Insert the new key-value pair*  *hashtable->array[index].key = key;*  *hashtable->array[index].value = value;*  *hashtable->array[index].status = 1; // Mark the slot as occupied*  *}*  *// Function to check if the array contains duplicate elements*  *int hasDuplicates(int arr[], int n) {*  *struct HashTable\* hashtable = createHashTable();*  *// Traverse the array and insert each element into the hash table*  *for (int i = 0; i < n; i++) {*  *int key = arr[i];*  *int value = i;*  *// If the element already exists in the hash table, it is a duplicate*  *if (hashtable->array[hashFunction1(key)].status == 1 &&*  *hashtable->array[hashFunction1(key)].key == key) {*  *free(hashtable->array);*  *free(hashtable);*  *return 1;*  *}*  *insert(hashtable, key, value);*  *}*  *free(hashtable->array);*  *free(hashtable);*  *return 0; // No duplicates found*  *}*  *int main() {*  *int arr[] = {1, 2, 3, 4, 5, 1};*  *int n = sizeof(arr) / sizeof(arr[0]);*  *if (hasDuplicates(arr, n)) {*  *printf("The array contains duplicate elements.\n");*  *} else {*  *printf("The array does not contain duplicate elements.\n");*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *1, 2, 3, 4, 5, 1* |
| **Sample Output:** |
| *The array contains duplicate elements.* |