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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | 02 | | |  |  | | | |
| **Student Name** | SAHANA KOLUR | | |  |  | | | |
| **SRN** | 01FE21BEC237 | | **Roll Number** | | 457 | **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.  C:\Users\Lenovo\Desktop\sign.png  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 Prefix Conversion | | | | | | | |
| 2.Duplicate Removal | | | | | | | |
| **In-order traversal of tree data structure** | 1.Expression Infix Conversion | | | | | | | |
| 2.Finding Kth Smallest Element | | | | | | | |
| **Post-order traversal of tree data structure** | 1.Evaluation of Arithmetic Expression Trees | | | | | | | |
| 2.Memory Management | | | | | | | |
| **DFS of graphs** | 1.Pathfinding | | | | | | | |
| 2.Cycle Detection | | | | | | | |
| **BFS of graphs** | 1.Social Networking | | | | | | | |
| 2.Puzzle solving | | | | | | | |
| **Linear probing of hashing** | 1.Spell Checking | | | | | | | |
| 2.Text Editors | | | | | | | |
| **Quadratic probing of hashing** | 1.Database Indexing | | | | | | | |
| 2.Data Deduplication | | | | | | | |
| **Double hashing** | 1.Cuckoo Hashing. | | | | | | | |
| 2.Network routing protocols. | | | | | | | |

|  |
| --- |
| **Approach:** Pre-order traversal of tree data structure |
| **Problem statement** |
| Prefix notation, also known as Polish notation, is a way of writing expressions where the operator comes before its operands. To convert an infix expression (the usual notation) to a prefix expression, you can use a binary expression tree. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAX\_EXPRESSION\_SIZE 100  typedef struct  {  char items[MAX\_EXPRESSION\_SIZE];  int top;  } Stack;  void initializeStack(Stack\* stack)  {  stack->top = -1;  }  int isStackEmpty(Stack\* stack)  {  return (stack->top == -1);  }  int isStackFull(Stack\* stack)  {  return (stack->top == MAX\_EXPRESSION\_SIZE - 1);  }  void push(Stack\* stack, char c)  {  if (!isStackFull(stack))  stack->items[++stack->top] = c;  }  char pop(Stack\* stack)  {  if (!isStackEmpty(stack))  return stack->items[stack->top--];  return '\0';  }  // Get the top element of the stack  char top(Stack\* stack) {  if (!isStackEmpty(stack))  return stack->items[stack->top];  return '\0';  }  int isOperator(char c)  {  return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');  }  int getPrecedence(char op)  {  if (op == '+' || op == '-')  return 1;  else if (op == '\*' || op == '/')  return 2;  else if (op == '^')  return 3;  return 0;  }  void infixToPrefix(char infix[], char prefix[])  {  Stack operatorStack, outputStack;  initializeStack(&operatorStack);  initializeStack(&outputStack);  int i, j = 0;  strrev(infix);  for (i = 0; infix[i] != '\0'; i++)  {  char token = infix[i];  if (isalnum(token))  {  prefix[j++] = token;  }  else if (token == ')')  {  push(&operatorStack, token);  }  else if (token == '(')  {  while (top(&operatorStack) != ')' && !isStackEmpty(&operatorStack))  {  prefix[j++] = pop(&operatorStack);  }  pop(&operatorStack);  }  else if (isOperator(token))  {  while (!isStackEmpty(&operatorStack) &&getPrecedence(top(&operatorStack)) >= getPrecedence(token))  {  prefix[j++] = pop(&operatorStack);  }  push(&operatorStack, token);  }  }  while (!isStackEmpty(&operatorStack))  {  prefix[j++] = pop(&operatorStack);  }  prefix[j] = '\0';  strrev(prefix);  }  int main()  {  char infixExpression[MAX\_EXPRESSION\_SIZE];  char prefixExpression[MAX\_EXPRESSION\_SIZE];  printf("Enter the infix expression: ");  fgets(infixExpression, sizeof(infixExpression), stdin);  infixExpression[strcspn(infixExpression, "\n")] = '\0';  infixToPrefix(infixExpression, prefixExpression);  printf("Prefix expression: %s\n", prefixExpression);  return 0;  } |
| **Sample Input:** |
| Enter the infix expression: a\*b+c/d |
| **Sample Output:** |
| Prefix expression: +\*ab/cd |

|  |
| --- |
| **Approach:** In-order traversal of tree data structure |
| **Problem statement** |
| To find the kth element during an inorder traversal of a binary tree, we can perform the traversal and keep track of the elements visited. When we reach the kth element, we can return it. |
| **Code** |
| #include <stdio.h>  void swap(int \*a, int \*b)  {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  int selectionSortKthSmallest(int arr[], int n, int k)  {  for (int i = 0; i< k; i++)  {  int min\_index = i;  for (int j = i + 1; j < n; j++)  {  if (arr[j] <arr[min\_index])  {  min\_index = j;  }  }  swap(&arr[i], &arr[min\_index]);  }  return arr[k - 1];  }  int main()  {  int n, k;  printf("Enter the size of the list: ");  scanf("%d", &n);  int arr[n];  printf("Enter %d elements:\n", n);  for (int i = 0; i< n; i++)  {  scanf("%d", &arr[i]);  }  printf("Enter the value of K: ");  scanf("%d", &k);  if (k > 0 && k <= n)  {  int kthSmallest = selectionSortKthSmallest(arr, n, k);  printf("The %dth smallest element is: %d\n", k, kthSmallest);  } else  {  printf("Invalid value of K. Please enter a value between 1 and %d.\n", n);  }  return 0;  } |
| **Sample Input:** |
| Enter the size of the list: 5  Enter 5 elements:  9 4 7 2 1  Enter the value of K: 3 |
| **Sample Output:** |
| The 3rd smallest element is: 4 |

|  |
| --- |
| **Approach:** post-order traversal of tree data structure |
| **Problem statement** |
| To evaluate an arithmetic expression tree using post-order traversal, we can recursively evaluate the left and right subtrees, and then perform the corresponding arithmetic operation at each node. |
| **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 isOperator(char c)  {  return (c == '+' || c == '-' || c == '\*' || c == '/');  }  int evaluateExpressionTree(struct Node\* root)  {  if (root == NULL)  {  return 0;  }  if (!isOperator(root->data))  {  return root->data - '0';  }  int leftValue = evaluateExpressionTree(root->left);  int rightValue = evaluateExpressionTree(root->right);  switch (root->data)  {  case '+': return leftValue + rightValue;  case '-': return leftValue - rightValue;  case '\*': return leftValue \* rightValue;  case '/': return leftValue / rightValue;  default: return 0;  }  }  struct Node\* buildExpressionTree(char postfix[])  {  int top = -1;  struct Node\* stack[100];  for (int i = 0; postfix[i] != '\0'; i++)  {  char c = postfix[i];  if (isdigit(c))  {  stack[++top] = createNode(c);  }  else if (isOperator(c))  {  struct Node\* newNode = createNode(c);  newNode->right = stack[top--];  newNode->left = stack[top--];  stack[++top] = newNode;  }  }  return stack[top];  }  int main()  {  char postfixExpression[100];  printf("Enter the postfix expression: ");  scanf("%s", postfixExpression);  struct Node\* root = buildExpressionTree(postfixExpression);  int result = evaluateExpressionTree(root);  printf("Result: %d\n", result);  return 0;  } |
| **Sample Input:** |
| Enter the postfix expression: 53+ |
| **Sample Output:** |
| Result: 8 |

|  |
| --- |
| **Approach:** DFS |
| **Problem statement** |
| In Depth-First Search (DFS), we can find paths from a source node to all other nodes in a graph. To do this, we can perform a recursive DFS traversal starting from the source node and keep track of the visited nodes and the path being formed during the traversal |
| **Code** |
| #include <stdio.h>  #include <stdbool.h>  #include <limits.h>  #define MAX\_NODES 100  struct Edge  {  int destination;  int weight;  };  int minDistance(int distance[], bool visited[], int nodes)  {  int min = INT\_MAX;  int min\_index;  for (int i = 0; i< nodes; i++)  {  if (!visited[i] && distance[i] <= min)  {  min = distance[i];  min\_index = i;  }  }  return min\_index;  }  void printPath(int parent[], int j)  {  if (parent[j] == -1)  {  printf("%d ", j);  return;  }  printPath(parent, parent[j]);  printf("%d ", j);  }  void printSolution(int distance[], int parent[], int src, int nodes)  {  printf("Node\tDistance\tPath\n");  for (int i = 0; i< nodes; i++)  {  printf("%d\t%d\t\t", i, distance[i]);  printPath(parent, i);  printf("\n");  }  }  void dijkstra(int graph[MAX\_NODES][MAX\_NODES], int src, int nodes)  {  int distance[MAX\_NODES];  bool visited[MAX\_NODES];  int parent[MAX\_NODES];  for (int i = 0; i< nodes; i++)  {  distance[i] = INT\_MAX;  visited[i] = false;  parent[i] = -1;  }  distance[src] = 0;  for (int count = 0; count < nodes - 1; count++)  {  int u = minDistance(distance, visited, nodes);  visited[u] = true;  for (int v = 0; v < nodes; v++)  {  if (!visited[v] && graph[u][v] && distance[u] != INT\_MAX &&distance[u] + graph[u][v] < distance[v])  {  distance[v] = distance[u] + graph[u][v];  parent[v] = u;  }  }  }  printSolution(distance, parent, src, nodes);  }  int main()  {  int nodes, edges;  int graph[MAX\_NODES][MAX\_NODES] = {0};  printf("Enter the number of nodes: ");  scanf("%d", &nodes);  printf("Enter the number of edges: ");  scanf("%d", &edges);  printf("Enter the edges and their weights (source destination weight):\n");  for (int i = 0; i< edges; i++)  {  int src, dest, weight;  scanf("%d %d %d", &src, &dest, &weight);  graph[src][dest] = weight;  }  int src, dest;  printf("Enter the source node: ");  scanf("%d", &src);  printf("Enter the destination node: ");  scanf("%d", &dest);  dijkstra(graph, src, nodes);  return 0;  } |
| **Sample Input:** |
| Enter the number of nodes: 6  Enter the number of edges: 9  Enter the edges and their weights (source destination weight):  0 1 5  0 2 3  1 3 6  1 2 2  2 4 4  2 5 2  2 3 7  3 4 1  4 5 3  Enter the source node: 0  Enter the destination node: 5 |
| **Sample Output:** |
| Node Distance Path  0 0 0  1 5 0 1  2 3 0 2  3 9 0 2 3  4 10 0 2 3 4  5 5 0 2 5 |

|  |
| --- |
| **Approach:** BFS |
| **Problem statement** |
| In Breadth-First Search (BFS) for puzzle solving, we can use a queue data structure to explore possible moves step by step until we find a solution. The puzzle state is represented as a node in the graph, and each node stores the current configuration of the puzzle |
| **Code** |
| #include <stdio.h>  #include <stdbool.h>  #define N 4  void printPuzzle(int puzzle[N][N])  {  for (int i = 0; i< N; i++)  {  for (int j = 0; j < N; j++)  {  printf("%d ", puzzle[i][j]);  }  printf("\n");  }  }  bool isSafe(int puzzle[N][N], int row, int col, int num)  {  for (int i = 0; i< N; i++)  {  if (puzzle[row][i] == num || puzzle[i][col] == num)  {  return false;  }  }  int startRow = row - (row % 2);  int startCol = col - (col % 2);  for (int i = 0; i< 2; i++)  {  for (int j = 0; j < 2; j++)  {  if (puzzle[startRow + i][startCol + j] == num)  {  return false;  }  }  }  return true;  }  bool solvePuzzle(int puzzle[N][N])  {  int row, col;  bool isEmpty = false;  for (row = 0; row < N; row++)  {  for (col = 0; col < N; col++)  {  if (puzzle[row][col] == 0)  {  isEmpty = true;  break;  }  }  if (isEmpty)  {  break;  }  }  if (!isEmpty)  {  return true;  }  for (int num = 1; num<= N; num++)  {  if (isSafe(puzzle, row, col, num))  {  puzzle[row][col] = num;  if (solvePuzzle(puzzle))  {  return true;  }  puzzle[row][col] = 0;  }  }  return false;  }  int main()  {  int puzzle[N][N];  printf("Enter the puzzle grid (4x4 matrix with 0 for empty cells):\n");  for (int i = 0; i< N; i++)  {  for (int j = 0; j < N; j++)  {  scanf("%d", &puzzle[i][j]);  }  }  if (solvePuzzle(puzzle))  {  printf("Solution:\n");  printPuzzle(puzzle);  }  else  {  printf("No solution found for the given puzzle.\n");  }  return 0;  } |
| **Sample Input:** |
| Enter the puzzle grid (4x4 matrix with 0 for empty cells):  0 0 3 0  1 0 0 2  0 2 0 0  0 0 1 0 |
| **Sample Output:** |
| 4 1 3 2  1 3 4 2  3 2 1 4  2 4 1 3 |

|  |
| --- |
| **Approach:** Linear probing |
| **Problem statement** |
| In linear probing, we can implement a simple spell-checking functionality by using a hash table to store a dictionary of words and then searching for user-input words in that hash table. When there is a collision in the hash table (two words with the same hash), we use linear probing to find the next available position to store the word. |
| **Code** |
| #include <stdio.h>  #include <stdbool.h>  #include <string.h>  #define MAX\_DICTIONARY\_WORDS 1000  #define MAX\_WORD\_LENGTH 50  bool isWordInDictionary(const char\* word, const char dictionary[][MAX\_WORD\_LENGTH], int dictionarySize)  {  for (int i = 0; i<dictionarySize; i++)  {  if (strcmp(word, dictionary[i]) == 0)  {  return true;  }  }  return false;  }  int main()  {  char dictionary[MAX\_DICTIONARY\_WORDS][MAX\_WORD\_LENGTH];  int dictionarySize = 0;  printf("Enter words for the dictionary (type 'stop' to finish):\n");  while (dictionarySize< MAX\_DICTIONARY\_WORDS)  {  char word[MAX\_WORD\_LENGTH];  scanf("%s", word);  if (strcmp(word, "stop") == 0)  {  break;  }  strncpy(dictionary[dictionarySize], word, MAX\_WORD\_LENGTH);  dictionarySize++;  }  while (getchar() != '\n');  printf("\nEnter a sentence for spell checking:\n");  char sentence[MAX\_WORD\_LENGTH \* MAX\_DICTIONARY\_WORDS];  fgets(sentence, sizeof(sentence), stdin);  char\* wordPtr = strtok(sentence, " \n");  while (wordPtr != NULL)  {  char word[MAX\_WORD\_LENGTH];  int i, j;  for (i = 0, j = 0; wordPtr[i] != '\0'; i++)  {  if (isalpha(wordPtr[i]))  {  word[j++] = tolower(wordPtr[i]);  }  }  word[j] = '\0';  if (!isWordInDictionary(word, dictionary, dictionarySize))  {  printf("Unknown word: %s\n", word);  }  wordPtr = strtok(NULL, " \n");  }  return 0;  } |
| **Sample Input:** |
| Enter words for the dictionary (type 'stop' to finish):  hello  world  programming  language  stop  Enter a sentence for spell checking:  Hello, I am learning programming languages. |
| **Sample Output:** |
| Unknown word: languages |

|  |
| --- |
| **Approach:** quadratic probing |
| **Problem statement** |
| In quadratic probing, we can implement database indexing by using a hash table to store records and handling collisions using quadratic probing. The hash table will store the records, and when a collision occurs, we will use a quadratic function to probe for the next available position to store the record. |
| **Code** |
| #include <stdio.h>  #include <string.h>  #include <stdbool.h>  #define MAX\_RECORDS 100  #define MAX\_NAME\_LENGTH 50  struct Record  {  char name[MAX\_NAME\_LENGTH];  int age;  };  struct Record\* searchRegular(struct Record database[], int numRecords, const char\* name)  {  for (int i = 0; i<numRecords; i++)  {  if (strcmp(database[i].name, name) == 0)  {  return &database[i];  }  }  return NULL;  }  struct Record\* searchIndexed(struct Record database[], struct Record\* index[], int numRecords, const char\* name)  {  int hash = name[0] % numRecords;  struct Record\* record = index[hash];  while (record != NULL)  {  if (strcmp(record->name, name) == 0)  {  return record;  }  record = record->name;  }  return NULL;  }  int main()  {  struct Record database[MAX\_RECORDS];  struct Record\* index[MAX\_RECORDS] = { NULL };  int numRecords = 0;  printf("Enter records to the database (type 'stop' for name to finish):\n");  while (numRecords< MAX\_RECORDS)  {  char name[MAX\_NAME\_LENGTH];  int age;  printf("Name: ");  scanf("%s", name);  if (strcmp(name, "stop") == 0)  {  break;  }  printf("Age: ");  scanf("%d", &age);  strcpy(database[numRecords].name, name);  database[numRecords].age = age;  int hash = name[0] % MAX\_RECORDS;  database[numRecords].name = index[hash];  index[hash] = &database[numRecords];  numRecords++;  }  char searchName[MAX\_NAME\_LENGTH];  printf("\nEnter a name to search: ");  scanf("%s", searchName);  struct Record\* foundRecord = searchRegular(database, numRecords, searchName);  if (foundRecord != NULL)  {  printf("Record found (regular search): %s, %d years old\n", foundRecord->name, foundRecord->age);  }  else  {  printf("Record not found (regular search).\n");  }  foundRecord = searchIndexed(database, index, numRecords, searchName);  if (foundRecord != NULL)  {  printf("Record found (indexed search): %s, %d years old\n", foundRecord->name, foundRecord->age);  }  else  {  printf("Record not found (indexed search).\n");  }  return 0;  } |
| **Sample Input:** |
| Enter records to the database (type 'stop' for name to finish):  Name: Alice  Age: 30  Name: Bob  Age: 25  Name: Carol  Age: 40  Name: stop  Enter a name to search: Carol |
| **Sample Output:** |
| Record found (regular search): Carol, 40 years old  Record found (indexed search): Carol, 40 years old |

|  |
| --- |
| **Approach:** double hashing |
| **Problem statement** |
| Cuckoo hashing is a hash table implementation that resolves collisions by using multiple hash functions and multiple tables. If a collision occurs during insertion, the existing item is "kicked out" to another table using a different hash function until a vacant spot is found. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define TABLE\_SIZE 10  typedef struct {  char destination[20];  int nextHop;  } RouteEntry;  int hash1(const char\* destination) {  int hash = 0;  for (int i = 0; destination[i] != '\0'; i++) {  hash += destination[i];  }  return hash % TABLE\_SIZE;  }  int hash2(const char\* destination) {  int hash = 0;  for (int i = 0; destination[i] != '\0'; i++) {  hash += destination[i] \* 31;  }  return (hash % (TABLE\_SIZE - 1)) + 1;  }  void insert(RouteEntry\* routingTable, const char\* destination, int nextHop) {  int hashIndex = hash1(destination);  int stepSize = hash2(destination);  while (routingTable[hashIndex].nextHop != -1) {  hashIndex = (hashIndex + stepSize) % TABLE\_SIZE;  }  strcpy(routingTable[hashIndex].destination, destination);  routingTable[hashIndex].nextHop = nextHop;  }  int search(RouteEntry\* routingTable, const char\* destination) {  int hashIndex = hash1(destination);  int stepSize = hash2(destination);  while (strcmp(routingTable[hashIndex].destination, destination) != 0) {  return -1;  }  hashIndex = (hashIndex + stepSize) % TABLE\_SIZE;  }  return routingTable[hashIndex].nextHop;  }  void initializeRoutingTable(RouteEntry\* routingTable) {  for (int i = 0; i< TABLE\_SIZE; i++) {  strcpy(routingTable[i].destination, "");  routingTable[i].nextHop = -1;  }  }  void displayRoutingTable(RouteEntry\* routingTable) {  printf("Routing Table:\n");  for (int i = 0; i< TABLE\_SIZE; i++) {  if (routingTable[i].nextHop != -1) {  printf("[%s] -> Next Hop: %d\n", routingTable[i].destination, routingTable[i].nextHop);  } else {  printf("[%s] -> NULL\n", routingTable[i].destination);  }  }  }  int main() {  RouteEntry\* routingTable = (RouteEntry\*)malloc(TABLE\_SIZE \* sizeof(RouteEntry));  initializeRoutingTable(routingTable);  int numEntries;  printf("Enter the number of route entries to insert: ");  scanf("%d", &numEntries);  getchar(); // Clear the newline character from the input buffer  for (int i = 0; i<numEntries; i++) {  char destination[20];  int nextHop;  printf("Enter the destination for route entry %d: ", i + 1);  fgets(destination, sizeof(destination), stdin);  destination[strcspn(destination, "\n")] = '\0'; // Remove the newline character  printf("Enter the next hop for route entry %d: ", i + 1);  scanf("%d", &nextHop);  getchar(); // Clear the newline character from the input buffer  insert(routingTable, destination, nextHop);  }  displayRoutingTable(routingTable);  char destinationToSearch[20];  printf("Enter the destination to search: ");  fgets(destinationToSearch, sizeof(destinationToSearch), stdin);  destinationToSearch[strcspn(destinationToSearch, "\n")] = '\0'; // Remove the newline character  int nextHop = search(routingTable, destinationToSearch);  if (nextHop != -1) {  printf("Next Hop for destination '%s': %d\n", destinationToSearch, nextHop);  } else {  printf("No route found for destination '%s'.\n", destinationToSearch);  }  free(routingTable);  return 0;  } |
| **Sample Input:** |
| Enter the number of route entries to insert: 4  Enter the destination for route entry 1: 192.168.0.0  Enter the next hop for route entry 1: 1  Enter the destination for route entry 2: 10.0.0.0  Enter the next hop for route entry 2: 2  Enter the destination for route entry 3: 172.16.0.0  Enter the next hop for route entry 3: 3  Enter the destination for route entry 4: 192.0.2.0  Enter the next hop for route entry 4: 4  Enter the destination to search: 10.0.0.0 |
| **Sample Output:** |
| Routing Table:  [192.168.0.0] -> Next Hop: 1  [10.0.0.0] -> Next Hop: 2  [172.16.0.0] -> Next Hop: 3  [192.0.2.0] -> Next Hop: 4  Next Hop for destination '10.0.0.0': 2 |