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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | 02 | | |  |  | | | |
| **Student Name** | Aishwarya B Kalatippi | | |  |  | | | |
| **SRN** | 01FE21BEI016 | | **Roll Number** | | 115 | **Division** | A[II] | |
| **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. Prefix expression evaluation | | | | | | | |
| 2. AI Decision Making. | | | | | | | |
| **In-order traversal of tree data structure** | 1.Finding the kth smallest element. | | | | | | | |
| 2. Huffman Encoding | | | | | | | |
| **Post-order traversal of tree data structure** | 1.Generating formula string. | | | | | | | |
| 2.Memory deallocation. | | | | | | | |
| **DFS of graphs** | 1. Generating permutations and combinations | | | | | | | |
| 2.Detecting cycles. | | | | | | | |
| **BFS of graphs** | 1.Web crawling. | | | | | | | |
| 2.Graph is cyclic or not. | | | | | | | |
| **Linear probing of hashing** | 1.Spelling Check. | | | | | | | |
| 2. Bloom Filters | | | | | | | |
| **Quadratic probing of hashing** | 1.Routines. | | | | | | | |
| 2. Data Integrity Verification | | | | | | | |
| **Double hashing** | 1. Compiler Symbol Tables | | | | | | | |
| 2. Perfect Hashing | | | | | | | |

|  |
| --- |
| **Approach:** Pre-order traversal of tree data structure |
| Problem statement : Prefix expression evaluation. |
| Prefix expression evaluation is best suited using pre-order traversal because the pre-order traversal follows a specific order of operations that aligns well with the evaluation of prefix expressions.  In pre-order traversal, the algorithm visits the current node before recursively visiting its left and right subtrees. This order matches the evaluation of prefix expressions, where the operator comes before its operands. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <string.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));  if (newNode == NULL) {  printf("Memory allocation failed!\n");  exit(1);  }  newNode->data = data;  newNode->left = newNode->right = NULL;  return newNode;  }  int isOperator(char c) {  return (c == '+' || c == '-' || c == '\*' || c == '/');  }  int evaluatePrefixExpression(struct Node\* root) {  if (root == NULL)  return 0;  if (!isOperator(root->data))  return root->data - '0';  int leftResult = evaluatePrefixExpression(root->left);  int rightResult = evaluatePrefixExpression(root->right);  switch (root->data) {  case '+':  return leftResult + rightResult;  case '-':  return leftResult - rightResult;  case '\*':  return leftResult \* rightResult;  case '/':  return leftResult / rightResult;  default:  return 0;  }  }  struct Node\* buildExpressionTree(char prefix[]) {  struct Node\*\* stack = (struct Node\*\*)malloc(strlen(prefix) \* sizeof(struct Node\*));  int top = -1;  for (int i = strlen(prefix) - 1; i >= 0; i--) {  char c = prefix[i];  if (isOperator(c)) {  struct Node\* newNode = createNode(c);  newNode->left = stack[top--];  newNode->right = stack[top--];  stack[++top] = newNode;  } else {  struct Node\* newNode = createNode(c);  stack[++top] = newNode;  }  }  struct Node\* root = stack[top--];  free(stack);  return root;  }  int main() {  char prefix[100];  printf("Enter the prefix expression: ");  fgets(prefix, sizeof(prefix), stdin);  prefix[strcspn(prefix, "\n")] = '\0';  struct Node\* root = buildExpressionTree(prefix);  int result = evaluatePrefixExpression(root);  printf("Result: %d\n", result);  return 0;  } |
| **Sample Input:** |
| Enter the prefix expression: \*+42+53 |
| **Sample Output:** |
| Result: 48 |

|  |
| --- |
| **Approach:** In-order traversal of tree data structure |
| Problem statement : Finding the kth smallest element. |
| kthSmallest function takes the root of the binary search tree and the value of k as input and returns the kth smallest element in the BST. The inorderTraversal function is used to perform the inorder traversal and keep track of the visited nodes. When the kth node is reached during the traversal, its value is stored in the result variable. The traversal is stopped once the kth smallest element is found, to avoid unnecessary traversals. |
| **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));  if (newNode == NULL) {  printf("Memory allocation failed!\n");  exit(1);  }  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 kthSmallestUtil(struct Node\* root, int k, int\* count, int\* result) {  if (root == NULL)  return;  kthSmallestUtil(root->left, k, count, result);  (\*count)++;  if (\*count == k) {  \*result = root->data;  return;  }  kthSmallestUtil(root->right, k, count, result);  }  int kthSmallest(struct Node\* root, int k) {  int count = 0;  int result = -1;  kthSmallestUtil(root, k, &count, &result);  return result;  }  int main() {  struct Node\* root = NULL;  int n, k;  printf("Enter the number of elements in the binary search tree: ");  scanf("%d", &n);  printf("Enter the elements: ");  for (int i = 0; i < n; i++) {  int data;  scanf("%d", &data);  root = insert(root, data);  }  printf("Enter the value of K: ");  scanf("%d", &k);  int kthSmallestElement = kthSmallest(root, k);  if (kthSmallestElement != -1)  printf("The %dth smallest element is: %d\n", k, kthSmallestElement);  else  printf("Invalid value of K.\n");  return 0;  } |
| **Sample Input:** |
| Enter the number of elements in the binary search tree: 5  Enter the elements: 10  23  45  89  1  Enter the value of K: 3 |
| **Sample Output:** |
| The 3th smallest element is: 23 |

|  |
| --- |
| **Approach:** post-order traversal of tree data structure |
| **Problem statement** |
| Post-order traversal can be employed to generate formula strings from expression trees. By visiting each node and recursively constructing the formula string based on the operators and operands, you can obtain a human-readable representation of the formula. |
| **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));  if (newNode == NULL) {  printf("Memory allocation failed!\n");  exit(1);  }  newNode->data = data;  newNode->left = newNode->right = NULL;  return newNode;  }  // Function for post-order traversal to generate formula string  void generateFormulaString(struct Node\* root) {  if (root == NULL)  return;  generateFormulaString(root->left);  generateFormulaString(root->right);  printf("%c ", root->data);  }  int main() {  struct Node\* root = createNode('+');  root->left = createNode('2');  root->right = createNode('\*');  root->right->left = createNode('3');  root->right->right = createNode('4');  printf("Formula string: ");  generateFormulaString(root);  printf("\n");  return 0;  } |
| **Sample Input:** |
| Enter the postfix expression:+2\*34 |
| **Sample Output:** |
| Formula string: 2 3 4 \* + |

|  |
| --- |
| **Approach: BFS** |
| **Problem statement :** Graph is cyclic or not. |
| To check whether a graph is cyclic or not using a Breadth-First Search (BFS) approach, you can maintain a queue to perform BFS traversal of the graph. During the traversal, you'll keep track of visited nodes and their parent nodes. If you encounter a node that is already visited and its parent is not the current node, it means there is a cycle in the graph. |
| **Code** |
| #include <stdio.h>  #include <stdbool.h>  #include <stdlib.h>  struct Node {  int vertex;  struct Node\* next;  };  struct Graph {  int V;  struct Node\*\* adj;  };  struct Node\* createNode(int v)  {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->vertex = v;  newNode->next = NULL;  return newNode;  }  struct Graph\* createGraph(int V)  {  struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));  graph->V = V;  graph->adj = (struct Node\*\*)malloc(V \* sizeof(struct Node\*));  for (int i = 0; i < V; i++)  graph->adj[i] = NULL;  return graph;  }  void addEdge(struct Graph\* graph, int v, int w)  {  struct Node\* newNode = createNode(w);  newNode->next = graph->adj[v];  graph->adj[v] = newNode;  newNode = createNode(v);  newNode->next = graph->adj[w];  graph->adj[w] = newNode;  }  bool isCyclicUtil(int v, bool visited[], int parent, struct Graph\* graph)  {  visited[v] = true;  struct Node\* node = graph->adj[v];  while (node != NULL) {  int adjacent = node->vertex;  if (!visited[adjacent]) {  if (isCyclicUtil(adjacent, visited, v, graph))  return true;  }  else if (adjacent != parent)  return true;  node = node->next;  }  return false;  }  bool isCyclic(struct Graph\* graph)  {  bool\* visited = (bool\*)malloc(graph->V \* sizeof(bool));  for (int i = 0; i < graph->V; i++)  visited[i] = false;  for (int u = 0; u < graph->V; u++) {  if (!visited[u])  if (isCyclicUtil(u, visited, -1, graph))  return true;  }  return false;  }  int main()  {  struct Graph\* g1 = createGraph(5);  addEdge(g1, 1, 0);  addEdge(g1, 0, 2);  addEdge(g1, 2, 1);  addEdge(g1, 0, 3);  addEdge(g1, 3, 4);  isCyclic(g1) ? printf("Graph contains cycle\n") : printf("Graph doesn't contain cycle\n");  struct Graph\* g2 = createGraph(3);  addEdge(g2, 0, 1);  addEdge(g2, 1, 2);  isCyclic(g2) ? printf("Graph contains cycle\n") : printf("Graph doesn't contain cycle\n");  return 0;  } |
| **Sample Input:** |
| 1 0  0 2  2 1  0 3  3 4 |
| **Sample Output:** |
| Graph is cyclic. |

|  |
| --- |
| Approach: DFS |
| **Problem statement :** Generating permutations and combinations |
| Generating permutations and combinations: DFS can generate permutations or combinations of a set of elements. By maintaining a visited array and recursively selecting and unselecting elements, you can generate all possible permutations or combinations. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  void generatePermutationsUtil(int\* nums, int\* visited, int\* permutation, int index, int n) {  if (index == n) {  for (int i = 0; i < n; i++) {  printf("%d ", permutation[i]);  }  printf("\n");  return;  }  for (int i = 0; i < n; i++) {  if (!visited[i]) {  visited[i] = 1;  permutation[index] = nums[i];  generatePermutationsUtil(nums, visited, permutation, index + 1, n);  visited[i] = 0;  }  }  }  void generatePermutations(int\* nums, int n) {  int\* visited = (int\*)calloc(n, sizeof(int));  int\* permutation = (int\*)malloc(n \* sizeof(int));  generatePermutationsUtil(nums, visited, permutation, 0, n);  free(visited);  free(permutation);  }  void generateCombinationsUtil(int\* nums, int\* combination, int index, int n, int k, int start) {  if (index == k) {  for (int i = 0; i < k; i++) {  printf("%d ", combination[i]);  }  printf("\n");  return;  }  for (int i = start; i < n; i++) {  combination[index] = nums[i];  generateCombinationsUtil(nums, combination, index + 1, n, k, i + 1);  }  }  void generateCombinations(int\* nums, int n, int k) {  int\* combination = (int\*)malloc(k \* sizeof(int));  generateCombinationsUtil(nums, combination, 0, n, k, 0);  free(combination);  }  int main() {  int n;  printf("Enter the number of elements: ");  scanf("%d", &n);  int\* nums = (int\*)malloc(n \* sizeof(int));  printf("Enter the elements: ");  for (int i = 0; i < n; i++) {  scanf("%d", &nums[i]);  }  printf("Permutations:\n");  generatePermutations(nums, n);  printf("Combinations:\n");  for (int k = 1; k <= n; k++) {  printf("Size %d:\n", k);  generateCombinations(nums, n, k);  }  free(nums);  return 0;  } |
| **Sample Input:** |
| Enter the number of elements: 3  Enter the elements: 1  2  1 |
| **Sample Output:** |
| Permutations:  1 2 1  1 1 2  2 1 1  2 1 1  1 1 2  1 2 1  Combinations:  Size 1:  1  2  1  Size 2:  1 2  1 1  2 1  Size 3:  1 2 1 |

|  |
| --- |
| **Approach:** Linear probing |
| **Problem statement :** **spell checker** |
| One straight-forward way to implement a fast spell checker is to use a hash table. If a word is found, it is assumed to be spelled correctly. Otherwise, it is assumed to be spelled incorrectly |
| **Code** |
| #include <stdio.h>  #include <string.h>  #include <stdbool.h>    #define TABLE\_SIZE 100    typedef struct {  char word[50];  bool isOccupied;  } Entry;    int hashFunction(char \*word) {  int sum = 0;  for (int i = 0; i < strlen(word); i++) {  sum += word[i];  }  return sum % TABLE\_SIZE;  }    void insertWord(Entry hashTable[], char \*word) {  int hashValue = hashFunction(word);  int probeCount = 0;  while (hashTable[hashValue].isOccupied) {  if (strcmp(hashTable[hashValue].word, word) == 0) {  printf("%s is already in the dictionary.\n", word);  return;  }  hashValue = (hashValue + 1) % TABLE\_SIZE;  probeCount++;  if (probeCount >= TABLE\_SIZE) {  printf("Hash table is full. Cannot insert %s.\n", word);  return;  }  }  strcpy(hashTable[hashValue].word, word);  hashTable[hashValue].isOccupied = true;  printf("%s inserted into the dictionary.\n", word);  }    bool searchWord(Entry hashTable[], char \*word) {  int hashValue = hashFunction(word);  int probeCount = 0;  while (hashTable[hashValue].isOccupied) {  if (strcmp(hashTable[hashValue].word, word) == 0) {  return true;  }  hashValue = (hashValue + 1) % TABLE\_SIZE;  probeCount++;  if (probeCount >= TABLE\_SIZE) {  break;  }  }  return false;  }    int main() {  Entry hashTable[TABLE\_SIZE];  for (int i = 0; i < TABLE\_SIZE; i++) {  hashTable[i].isOccupied = false;  }    char word[50];  char input[10];    printf("Spelling check program\n");  printf("----------------------\n");    while (1) {  printf("\nEnter a word (or 'q' to quit): ");  scanf("%s", input);  if (strcmp(input, "q") == 0) {  break;  }  strcpy(word, input);  insertWord(hashTable, word);  }    printf("\nSpelling check:\n");  while (1) {  printf("\nEnter a word to check (or 'q' to quit): ");  scanf("%s", input);  if (strcmp(input, "q") == 0) {  break;  }  strcpy(word, input);  printf("Is '%s' in the dictionary? %s\n", word, searchWord(hashTable, word) ? "Yes" : "No");  }    return 0;  } |
| **Sample Input:** |
| Spelling check program  ----------------------  Enter a word (or 'q' to quit): abcd  abcd inserted into the dictionary.  Enter a word (or 'q' to quit): pqrs  pqrs inserted into the dictionary.  Enter a word (or 'q' to quit): q |
| **Sample Output:** |
| Spelling check:  Enter a word to check (or 'q' to quit): abcc  Is 'abcc' in the dictionary? No  Enter a word to check (or 'q' to quit): pqrs  Is 'pqrs' in the dictionary? Yes  Enter a word to check (or 'q' to quit): q |

|  |
| --- |
| **Approach:** quadratic probing |
| **Problem statement : Routines** |
| Quadratic hashing is another technique used in hash tables to handle collisions. In quadratic hashing, a quadratic probing sequence is used to find an available slot when there is a collision. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>    #define SIZE 10    // Hash table structure  struct HashTable {  int data[SIZE];  int status[SIZE]; // 0 for empty, 1 for occupied, -1 for deleted  };    // Hash function  int hash(int key) {  return key % SIZE;  }    // Insert a value into the hash table  void insert(struct HashTable \*hashTable, int value) {  int index = hash(value);  int i = 1;    while (hashTable->status[index] == 1) {  index = (index + i \* i) % SIZE;  i++;  }    hashTable->data[index] = value;  hashTable->status[index] = 1;  }    // Search for a value in the hash table  int search(struct HashTable hashTable, int value) {  int index = hash(value);  int i = 1;    while (hashTable.status[index] != 0) {  if (hashTable.status[index] == 1 && hashTable.data[index] == value)  return index;    index = (index + i \* i) % SIZE;  i++;  }    return -1; // Value not found  }    // Delete a value from the hash table  void delete(struct HashTable \*hashTable, int value) {  int index = search(\*hashTable, value);    if (index != -1) {  hashTable->status[index] = -1;  printf("%d deleted from the hash table.\n", value);  } else {  printf("%d not found in the hash table.\n", value);  }  }    // Display the contents of the hash table  void display(struct HashTable hashTable) {  printf("Hash Table:\n");  for (int i = 0; i < SIZE; i++) {  if (hashTable.status[i] == 1)  printf("Index %d: %d\n", i, hashTable.data[i]);  else  printf("Index %d: Empty\n", i);  }  }    int main() {  struct HashTable hashTable;    // Initialize hash table  for (int i = 0; i < SIZE; i++)  hashTable.status[i] = 0;    int choice, value;    do {  printf("\n======= Hash Table Operations =======\n");  printf("1. Insert a value\n");  printf("2. Search a value\n");  printf("3. Delete a value\n");  printf("4. Display the hash table\n");  printf("5. Exit\n");  printf("Enter your choice: ");  scanf("%d", &choice);    switch (choice) {  case 1:  printf("Enter the value to insert: ");  scanf("%d", &value);  insert(&hashTable, value);  printf("%d inserted into the hash table.\n", value);  break;  case 2:  printf("Enter the value to search: ");  scanf("%d", &value);  int index = search(hashTable, value);  if (index != -1)  printf("%d found at index %d.\n", value, index);  else  printf("%d not found in the hash table.\n", value);  break;  case 3:  printf("Enter the value to delete: ");  scanf("%d", &value);  delete(&hashTable, value);  break;  case 4:  display(hashTable);  break;  case 5:  printf("Exiting the program.\n");  exit(0);  default:  printf("Invalid choice! Please try again.\n");  }  } while (1);    return 0;  } |
| **Sample Input:** |
| ======= Hash Table Operations =======  1. Insert a value  2. Search a value  3. Delete a value  4. Display the hash table  5. Exit  Enter your choice: 1  Enter the value to insert: 10  10 inserted into the hash table.  ======= Hash Table Operations =======  1. Insert a value  2. Search a value  3. Delete a value  4. Display the hash table  5. Exit  Enter your choice: 1  Enter the value to insert: 20  20 inserted into the hash table.  ======= Hash Table Operations =======  1. Insert a value  2. Search a value  3. Delete a value  4. Display the hash table  5. Exit  Enter your choice: 1  Enter the value to insert: 45  45 inserted into the hash table.  ======= Hash Table Operations =======  1. Insert a value  2. Search a value  3. Delete a value  4. Display the hash table  5. Exit  Enter your choice: 1  Enter the value to insert: 23  23 inserted into the hash table.  ======= Hash Table Operations =======  1. Insert a value  2. Search a value  3. Delete a value  4. Display the hash table  5. Exit  Enter your choice: 1  Enter the value to insert: 89  89 inserted into the hash table.  ======= Hash Table Operations =======  1. Insert a value  2. Search a value  3. Delete a value  4. Display the hash table  5. Exit |
| **Sample Output:** |
| Enter your choice: 4  Hash Table:  Index 0: 10  Index 1: 20  Index 2: Empty  Index 3: 23  Index 4: Empty  Index 5: 45  Index 6: Empty  Index 7: Empty  Index 8: Empty  Index 9: 89 |

|  |
| --- |
| **Approach:**double hashing |
| Problem statement : Hash Table |
| A symbol table can be implemented using a hash table, where each entry in the table corresponds to a symbol and its associated information. Each entry typically includes the symbol name, data type, scope information, memory address, and other relevant attributes. |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #define HASH\_TABLE\_SIZE 10  // Structure to represent a node in the linked list  struct ListNode {  int key;  int value;  struct ListNode\* next;  };  // Structure to represent the hash table  struct HashTable {  struct ListNode\* buckets[HASH\_TABLE\_SIZE];  };  // Hash function to calculate the bucket index  int hash(int key) {  return key % HASH\_TABLE\_SIZE;  }  // Function to create a new node for the linked list  struct ListNode\* createNode(int key, int value) {  struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));  newNode->key = key;  newNode->value = value;  newNode->next = NULL;  return newNode;  }  // Function to insert a key-value pair into the hash table  void insert(struct HashTable\* ht, int key, int value) {  int index = hash(key);  struct ListNode\* newNode = createNode(key, value);  if (ht->buckets[index] == NULL) {  ht->buckets[index] = newNode;  } else {  // Collision handling using separate chaining  struct ListNode\* current = ht->buckets[index];  while (current->next != NULL) {  current = current->next;  }  current->next = newNode;  }  }  // Function to search for a key in the hash table and return its value  int get(struct HashTable\* ht, int key) {  int index = hash(key);  struct ListNode\* current = ht->buckets[index];  while (current != NULL) {  if (current->key == key) {  return current->value;  }  current = current->next;  }  return -1; // Key not found  }  // Function to display the contents of the hash table  void displayHashTable(struct HashTable\* ht) {  printf("Hash Table:\n");  for (int i = 0; i < HASH\_TABLE\_SIZE; i++) {  printf("%d:", i);  struct ListNode\* current = ht->buckets[i];  while (current != NULL) {  printf(" -> (%d, %d)", current->key, current->value);  current = current->next;  }  printf("\n");  }  }  int main() {  struct HashTable ht;  for (int i = 0; i < HASH\_TABLE\_SIZE; i++) {  ht.buckets[i] = NULL;  }  // Insert key-value pairs into the hash table  insert(&ht, 5, 100);  insert(&ht, 15, 200);  insert(&ht, 25, 300);  insert(&ht, 7, 400);  insert(&ht, 17, 500);  // Display the contents of the hash table  displayHashTable(&ht);  // Search for a key in the hash table  int searchKey = 15;  int searchResult = get(&ht, searchKey);  if (searchResult != -1) {  printf("Value for key %d: %d\n", searchKey, searchResult);  } else {  printf("Key %d not found in the hash table.\n", searchKey);  }  return 0;  } |
| **Sample Input:** |
| Key: 5, Value: 100  Key: 15, Value: 200  Key: 25, Value: 300  Key: 7, Value: 400  Key: 17, Value: 500 |
| **Sample Output:** |
| Hash Table:  0:  1:  2:  3:  4:  5: -> (5, 100)  6:  7: -> (7, 400)  8:  9: -> (15, 200) -> (25, 300) -> (17, 500)  Value for key 15: 200 |