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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *02* | | |  |  | | | |
| **Student Name** | Bhushan S Sonar | | |  |  | | | |
| **SRN** | 01FE21bEC312 | | **Roll Number** | | 561 | **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.Printing the tree | | | | | | | |
| **In-order traversal of tree data structure** | 1.Expression parsing | | | | | | | |
| 2.Sorting | | | | | | | |
| **Post-order traversal of tree data structure** | 1. File system cleanup | | | | | | | |
| 2.Postfix expression conversion | | | | | | | |
| **DFS of graphs** | 1.Solving puzzles and games | | | | | | | |
| 2.Graph Connectivity | | | | | | | |
| **BFS of graphs** | 1.Shortest path | | | | | | | |
| 2.minimal spanning | | | | | | | |
| **Linear probing of hashing** | 1.Caching | | | | | | | |
| 2.Symbol tables | | | | | | | |
| **Quadratic probing of hashing** | 1.spell checking | | | | | | | |
| 2.file systems | | | | | | | |
| **Double hashing** | 1.Memory allocation | | | | | | | |
| 2.Load balancing | | | | | | | |

|  |
| --- |
| **Approach:** *Pre-order traversal of tree data structure* |
| **Problem statement** |
| Expression evaluation:An arithmetic expression in a space-separated pre-order format, where operators and operands are separated by spaces. Your task is to evaluate the expression and return the numerical result |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <stdbool.h>*  *#include <string.h>*  *struct TreeNode {*  *char val;*  *struct TreeNode\* left;*  *struct TreeNode\* right;*  *};*  *bool is\_operator(char c) {*  *return c == '+' || c == '-' || c == '\*' || c == '/';*  *}*  *struct TreeNode\* create\_node(char value) {*  *struct TreeNode\* new\_node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));*  *new\_node->val = value;*  *new\_node->left = NULL;*  *new\_node->right = NULL;*  *return new\_node;*  *}*  *struct TreeNode\* construct\_expression\_tree(char\* preorder) {*  *if (\*preorder == '\0') {*  *return NULL;*  *}*  *char token = \*preorder;*  *preorder++;*  *if (isdigit(token) || (token == '-' && isdigit(\*preorder))) {*  *return create\_node(token);*  *} else if (is\_operator(token)) {*  *struct TreeNode\* node = create\_node(token);*  *node->left = construct\_expression\_tree(preorder);*  *node->right = construct\_expression\_tree(preorder);*  *return node;*  *} else {*  *return NULL;*  *}*  *}*  *int evaluate\_expression\_tree(struct TreeNode\* root) {*  *if (!root) {*  *return 0;*  *}*  *if (!root->left && !root->right) {*  *return root->val - '0'; // Convert char to int*  *}*  *int left\_result = evaluate\_expression\_tree(root->left);*  *int right\_result = evaluate\_expression\_tree(root->right);*  *switch (root->val) {*  *case '+':*  *return left\_result + right\_result;*  *case '-':*  *return left\_result - right\_result;*  *case '\*':*  *return left\_result \* right\_result;*  *case '/':*  *return left\_result / right\_result; // Integer division*  *default:*  *return 0;*  *}*  *}*  *int evaluate\_expression(char\* expression) {*  *struct TreeNode\* root = construct\_expression\_tree(expression);*  *return evaluate\_expression\_tree(root);*  *}*  *int main() {*  *char expression[] = "+ \* 5 4 3"; // The expression is "5 + (4 \* 3)"*  *int result = evaluate\_expression(expression);*  *printf("Result: %d\n", result); // Output: 17*  *return 0;*  *}* |
| **Sample Input:** |
| *+ \* 5 4 3* |
| **Sample Output:** |
| *17* |

|  |
| --- |
| **Approach: In order traversal** |
| **Problem statement**. |
| Sorting:In an array of integers, and your task is to sort the elements in ascending order using the in-order traversal of a binary search tree |
| **Code**  #include <stdio.h>  #include <stdlib.h>  struct TreeNode {  int val;  struct TreeNode\* left;  struct TreeNode\* right;  };  struct TreeNode\* createNode(int value) {  struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));  newNode->val = value;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  struct TreeNode\* insert(struct TreeNode\* root, int value) {  if (root == NULL) {  return createNode(value);  }  if (value < root->val) {  root->left = insert(root->left, value);  } else if (value > root->val) {  root->right = insert(root->right, value);  }  return root;  }  void inOrderTraversal(struct TreeNode\* root) {  if (root != NULL) {  inOrderTraversal(root->left);  printf("%d ", root->val);  inOrderTraversal(root->right);  }  }  void freeTree(struct TreeNode\* root) {  if (root != NULL) {  freeTree(root->left);  freeTree(root->right);  free(root);  }  }  int main() {  int arr[] = {5, 3, 8, 2, 4, 7, 9, 1, 6};  int n = sizeof(arr) / sizeof(arr[0]);  struct TreeNode\* root = NULL;  // Construct the binary search tree  for (int i = 0; i < n; i++) {  root = insert(root, arr[i]);  }  // Perform in-order traversal for sorting  printf("Sorted elements: ");  inOrderTraversal(root);  printf("\n");  // Free the memory used for the tree  freeTree(root);  return 0;  } |
| **Sample Input:** |
| *5,4,7,1,2,9,3,6,8* |
| **Sample Output:** |
| *1,2,3,4,5,6,7,8,9* |

Note: Replicate the table for 7 more times (for each application- 1 table)

|  |
| --- |
| **Approach:**  *post order traversal of tress* |
| **Problem statement** |
| *File system cleanup: Post-order traversal can be used to delete files and directories in a file system, ensuring that all subdirectories and their contents are deleted before deleting the parent directory* |
| **Code** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <dirent.h>*  *#include <string.h>*  *// Function to delete a file or directory*  *void deleteFileOrDirectory(const char \*path) {*  *int status = remove(path);*  *if (status == 0) {*  *printf("Deleted: %s\n", path);*  *} else {*  *printf("Failed to delete: %s\n", path);*  *}*  *}*  *// Function to perform post-order traversal and delete files and directories*  *void postOrderCleanup(const char \*directoryPath) {*  *DIR \*dir;*  *struct dirent \*entry;*  *char path[256];*  *dir = opendir(directoryPath);*  *if (dir == NULL) {*  *perror("Error opening directory");*  *return;*  *}*  *while ((entry = readdir(dir)) != NULL) {*  *if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0) {*  *continue;*  *}*  *snprintf(path, sizeof(path), "%s/%s", directoryPath, entry->d\_name);*  *if (entry->d\_type == DT\_DIR) {*  *postOrderCleanup(path); // Recursively delete subdirectories*  *} else {*  *deleteFileOrDirectory(path); // Delete individual files*  *}*  *}*  *closedir(dir);*  *// Delete the directory itself after its contents have been deleted*  *deleteFileOrDirectory(directoryPath);*  *}*  *int main() {*  *// Sample input 1: Cleanup "sample\_directory\_1"*  *postOrderCleanup("sample\_directory\_1");*  *// Sample input 2: Cleanup "sample\_directory\_2"*  *postOrderCleanup("sample\_directory\_2");*  *return 0;*  *}* |
| **Sample Input:** |
| *sample\_directory\_2/*  *├── subdirectory1/*  *│ ├── subsubdirectory1/*  *│ │ └── file1.txt*  *│ └── subsubdirectory2/*  *│ └── file2.txt*  *└── subdirectory2/*  *└── file3.txt* |
| **Sample Output:** |
| *Deleted: file1.txt*  *Deleted: subsubdirectory1*  *Deleted: file2.txt*  *Deleted: subsubdirectory2*  *Deleted: subdirectory1*  *Deleted: file3.txt*  *Deleted: subdirectory2*  *Deleted: sample\_directory\_2* |

|  |
| --- |
| **Approach:** *DFS* |
| **Problem statement** |
| *Puzzle and games:* You are tasked with solving the classic "N-Queens" puzzle. Given an N×N chessboard, the N-Queens puzzle requires placing N queens on the board in such a way that no two queens attack each other. In chess, a queen can move horizontally, vertically, and diagonally. |
| **Code** |
| #include <stdio.h>  #include <stdbool.h>  #define N 4 // Change N to the desired board size  bool isSafe(int board[N][N], int row, int col) {  int i, j;  // Check if there is a queen in the same row on the left side  for (i = 0; i < col; i++) {  if (board[row][i])  return false;  }  // Check if there is a queen in the upper diagonal on the left side  for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {  if (board[i][j])  return false;  }  // Check if there is a queen in the lower diagonal on the left side  for (i = row, j = col; j >= 0 && i < N; i++, j--) {  if (board[i][j])  return false;  }  return true;  }  bool solveNQueensUtil(int board[N][N], int col) {  if (col >= N)  return true;  for (int i = 0; i < N; i++) {  if (isSafe(board, i, col)) {  board[i][col] = 1;  if (solveNQueensUtil(board, col + 1))  return true;  board[i][col] = 0; // Backtrack if the current placement doesn't lead to a solution  }  }  return false;  }  bool solveNQueens() {  int board[N][N] = {0};  if (solveNQueensUtil(board, 0) == false) {  printf("No solution exists for %d-Queens puzzle.\n", N);  return false;  }  // Print the solution  printf("Solution for %d-Queens puzzle:\n", N);  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++)  printf("%d ", board[i][j]);  printf("\n");  }  return true;  }  int main() {  solveNQueens();  return 0;  } |
| **Sample Input:** |
| *N=4* |
| **Sample Output:** |
| *Solution for 4-Queens puzzle:*  *0 1 0 0*  *0 0 0 1*  *1 0 0 0*  *0 0 1 0* |

|  |
| --- |
| **Approach:** *BFS* |
| **Problem statement** |
| *Shortest path:* You are given a binary tree, and your task is to find the shortest path between two nodes in the tree using Breadth-First Search (BFS). |
| **Code** |
| #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  struct TreeNode {  int val;  struct TreeNode\* left;  struct TreeNode\* right;  };  struct TreeNode\* createNode(int value) {  struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));  newNode->val = value;  newNode->left = NULL;  newNode->right = NULL;  return newNode;  }  struct QueueNode {  struct TreeNode\* node;  struct QueueNode\* next;  };  struct Queue {  struct QueueNode\* front;  struct QueueNode\* rear;  };  struct Queue\* createQueue() {  struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));  queue->front = NULL;  queue->rear = NULL;  return queue;  }  void enqueue(struct Queue\* queue, struct TreeNode\* node) {  struct QueueNode\* newNode = (struct QueueNode\*)malloc(sizeof(struct QueueNode));  newNode->node = node;  newNode->next = NULL;  if (queue->front == NULL) {  queue->front = newNode;  queue->rear = newNode;  } else {  queue->rear->next = newNode;  queue->rear = newNode;  }  }  struct TreeNode\* dequeue(struct Queue\* queue) {  if (queue->front == NULL)  return NULL;  struct TreeNode\* dequeuedNode = queue->front->node;  struct QueueNode\* temp = queue->front;  queue->front = queue->front->next;  free(temp);  return dequeuedNode;  }  void freeQueue(struct Queue\* queue) {  while (queue->front != NULL) {  dequeue(queue);  }  free(queue);  }  struct TreeNode\* findNode(struct TreeNode\* root, int target) {  if (root == NULL)  return NULL;  if (root->val == target)  return root;  struct TreeNode\* leftResult = findNode(root->left, target);  if (leftResult != NULL)  return leftResult;  struct TreeNode\* rightResult = findNode(root->right, target);  return rightResult;  }  int findShortestPath(struct TreeNode\* root, int source, int destination) {  if (root == NULL)  return -1;  if (source == destination)  return 0;  struct Queue\* queue = createQueue();  enqueue(queue, root);  int distance = 0;  bool destinationFound = false;  while (!destinationFound && queue->front != NULL) {  int levelSize = queue->front->next == NULL ? 1 : queue->rear - queue->front + 1;  for (int i = 0; i < levelSize; i++) {  struct TreeNode\* currentNode = dequeue(queue);  if (currentNode->val == destination) {  destinationFound = true;  break;  }  if (currentNode->left != NULL)  enqueue(queue, currentNode->left);  if (currentNode->right != NULL)  enqueue(queue, currentNode->right);  }  distance++;  }  freeQueue(queue);  if (destinationFound)  return distance - 1; // Subtract 1 as we count the edges, not nodes  else  return -1;  }  int main() {  struct TreeNode\* root = createNode(1);  root->left = createNode(2);  root->right = createNode(3);  root->left->left = createNode(4);  root->left->right = createNode(5);  root->right->right = createNode(6);  root->right->right->left = createNode(7);  root->right->right->right = createNode(8);  int source = 2;  int destination = 8;  int shortestPath = findShortestPath(root, source, destination);  if (shortestPath != -1)  printf("Shortest path from node %d to node %d is %d\n", source, destination, shortestPath);  else  printf("There is no path from node %d to node %d\n", source, destination);  return 0;  } |
| **Sample Input:** |
| *Binary Tree:*  *1*  */ \*  *2 3*  */ \ \*  *4 5 6*  */ \*  *7 8*  *Source Node: 2*  *Destination Node: 8* |
| **Sample Output:** |
| *v* *Shortest path from node 2 to node 8 is 3* |

|  |
| --- |
| **Approach:**  *Linear probe of hashing* |
| **Problem statement** |
| *Caching:* You are tasked with implementing a simple cache using linear probing of hashing. The cache should support insertion and retrieval of key-value pairs and handle collisions using linear probing. |
| **Code** |
| *#include <stdio.h>*  *#include <stdbool.h>*  *#include <stdlib.h>*  *#define TABLE\_SIZE 10*  *struct CacheEntry {*  *int key;*  *int value;*  *bool isValid;*  *};*  *struct Cache {*  *struct CacheEntry table[TABLE\_SIZE];*  *};*  *// Hash function using modulo to get the index in the hash table*  *int hashFunction(int key) {*  *return key % TABLE\_SIZE;*  *}*  *void initializeCache(struct Cache\* cache) {*  *for (int i = 0; i < TABLE\_SIZE; i++) {*  *cache->table[i].isValid = false;*  *}*  *}*  *void insert(struct Cache\* cache, int key, int value) {*  *int index = hashFunction(key);*  *int i = 1; // Linear probing step*  *while (cache->table[index].isValid) {*  *index = (index + i) % TABLE\_SIZE;*  *i++;*  *}*  *cache->table[index].key = key;*  *cache->table[index].value = value;*  *cache->table[index].isValid = true;*  *}*  *int search(struct Cache\* cache, int key) {*  *int index = hashFunction(key);*  *int i = 1; // Linear probing step*  *while (cache->table[index].isValid) {*  *if (cache->table[index].key == key)*  *return cache->table[index].value;*  *index = (index + i) % TABLE\_SIZE;*  *i++;*  *}*  *return -1; // Key not found*  *}*  *int main() {*  *struct Cache cache;*  *initializeCache(&cache);*  *// Insert key-value pairs into the cache*  *insert(&cache, 3, 10);*  *insert(&cache, 15, 25);*  *insert(&cache, 9, 99);*  *// Search for values using keys*  *int value1 = search(&cache, 3);*  *int value2 = search(&cache, 15);*  *int value3 = search(&cache, 9);*  *int value4 = search(&cache, 5);*  *printf("Value for key 3: %d\n", value1); // Output: 10*  *printf("Value for key 15: %d\n", value2); // Output: 25*  *printf("Value for key 9: %d\n", value3); // Output: 99*  *printf("Value for key 5: %d\n", value4); // Output: -1 (Key not found)*  *return 0;*  *}* |
| **Sample Input:** |
| *Cache Size: 10*  *Insertions:*  *(3, 10)*  *(15, 25)*  *(9, 99)*  *Retrievals:*  *Key: 3*  *Key: 15*  *Key: 9*  *Key: 5* |
| **Sample Output:** |
| *Value for key 3: 10*  *Value for key 15: 25*  *Value for key 9: 99*  *Value for key 5: -1* |

|  |
| --- |
| **Approach:** **Quadratic probing of hashing** |
| **Problem statement** |
| Spell checking: You are tasked with implementing a simple spell checker using quadratic probing of hashing. The spell checker should store a dictionary of valid words in a hash table and check if a given word exists in the dictionary. |
| **Code** |
| *#include <stdio.h>*  *#include <stdbool.h>*  *#include <stdlib.h>*  *#include <string.h>*  *#define TABLE\_SIZE 100*  *#define MAX\_WORD\_LENGTH 50*  *struct DictionaryEntry {*  *char word[MAX\_WORD\_LENGTH];*  *bool isValid;*  *};*  *struct Dictionary {*  *struct DictionaryEntry table[TABLE\_SIZE];*  *};*  *// Hash function using the first character of the word*  *int hashFunction(char\* word) {*  *return word[0] % TABLE\_SIZE;*  *}*  *void initializeDictionary(struct Dictionary\* dictionary) {*  *for (int i = 0; i < TABLE\_SIZE; i++) {*  *dictionary->table[i].isValid = false;*  *}*  *}*  *void insertWord(struct Dictionary\* dictionary, char\* word) {*  *int index = hashFunction(word);*  *int i = 0;*  *int step = 1;*  *while (dictionary->table[index].isValid) {*  *if (strcmp(dictionary->table[index].word, word) == 0) {*  *// Word already exists in the dictionary, no need to insert*  *return;*  *}*  *// Quadratic probing*  *index = (index + step \* i) % TABLE\_SIZE;*  *i++;*  *}*  *// Insert the word at the found index*  *strcpy(dictionary->table[index].word, word);*  *dictionary->table[index].isValid = true;*  *}*  *bool checkWord(struct Dictionary\* dictionary, char\* word) {*  *int index = hashFunction(word);*  *int i = 0;*  *int step = 1;*  *while (dictionary->table[index].isValid) {*  *if (strcmp(dictionary->table[index].word, word) == 0) {*  *// Word found in the dictionary*  *return true;*  *}*  *// Quadratic probing*  *index = (index + step \* i) % TABLE\_SIZE;*  *i++;*  *}*  *// Word not found in the dictionary*  *return false;*  *}*  *int main() {*  *struct Dictionary dictionary;*  *initializeDictionary(&dictionary);*  *// Populate the dictionary with some valid words*  *insertWord(&dictionary, "apple");*  *insertWord(&dictionary, "banana");*  *insertWord(&dictionary, "orange");*  *insertWord(&dictionary, "grape");*  *insertWord(&dictionary, "pear");*  *// Spell check some words*  *char word1[] = "apple";*  *char word2[] = "grape";*  *char word3[] = "banana";*  *char word4[] = "cherry";*  *printf("%s is %s in the dictionary.\n", word1, checkWord(&dictionary, word1) ? "found" : "not found");*  *printf("%s is %s in the dictionary.\n", word2, checkWord(&dictionary, word2) ? "found" : "not found");*  *printf("%s is %s in the dictionary.\n", word3, checkWord(&dictionary, word3) ? "found" : "not found");*  *printf("%s is %s in the dictionary.\n", word4, checkWord(&dictionary, word4) ? "found" : "not found");*  *return 0;*  *}* |
| **Sample Input:** |
| *Dictionary Size: 100*  *Insertions (Some valid words to populate the dictionary):*  *- apple*  *- banana*  *- orange*  *- grape*  *- pear*  *Spell Check:*  *- apple*  *- grape*  *- banana*  *- cherry* |
| **Sample Output:** |
| *apple is found in the dictionary.*  *grape is found in the dictionary.*  *banana is found in the dictionary.*  *cherry is not found in the dictionary.* |

|  |
| --- |
| **Approach:** **Double hashing** |
| **Problem statement** |
| *Load balancing:* You are tasked with implementing a load balancing system using double hashing as a collision resolution technique. The load balancing system is designed to distribute tasks across multiple processing units efficiently. |
| **Code** |
| *#include <stdio.h>*  *#include <stdbool.h>*  *#include <stdlib.h>*  *#include <string.h>*  *#define TABLE\_SIZE 10*  *#define MAX\_TASK\_LENGTH 50*  *struct TaskEntry {*  *char task[MAX\_TASK\_LENGTH];*  *bool isValid;*  *};*  *struct LoadBalancer {*  *struct TaskEntry table[TABLE\_SIZE];*  *};*  *// First hash function using the length of the task*  *int hashFunction1(char\* task) {*  *int length = strlen(task);*  *return length % TABLE\_SIZE;*  *}*  *// Second hash function using the ASCII sum of characters in the task*  *int hashFunction2(char\* task) {*  *int sum = 0;*  *int len = strlen(task);*  *for (int i = 0; i < len; i++) {*  *sum += (int)task[i];*  *}*  *return sum % TABLE\_SIZE;*  *}*  *void initializeLoadBalancer(struct LoadBalancer\* lb) {*  *for (int i = 0; i < TABLE\_SIZE; i++) {*  *lb->table[i].isValid = false;*  *}*  *}*  *void insertTask(struct LoadBalancer\* lb, char\* task) {*  *int index = hashFunction1(task);*  *int i = 0;*  *int step = hashFunction2(task);*  *while (lb->table[index].isValid) {*  *// Handle duplicate tasks*  *if (strcmp(lb->table[index].task, task) == 0) {*  *printf("Task '%s' already exists in the load balancer.\n", task);*  *return;*  *}*  *// Double hashing*  *index = (index + i \* step) % TABLE\_SIZE;*  *i++;*  *}*  *// Insert the task at the found index*  *strcpy(lb->table[index].task, task);*  *lb->table[index].isValid = true;*  *printf("Task '%s' added to the load balancer.\n", task);*  *}*  *void removeTask(struct LoadBalancer\* lb, char\* task) {*  *int index = hashFunction1(task);*  *int i = 0;*  *int step = hashFunction2(task);*  *while (lb->table[index].isValid) {*  *if (strcmp(lb->table[index].task, task) == 0) {*  *// Task found, remove it*  *lb->table[index].isValid = false;*  *printf("Task '%s' removed from the load balancer.\n", task);*  *return;*  *}*  *// Double hashing*  *index = (index + i \* step) % TABLE\_SIZE;*  *i++;*  *}*  *printf("Task '%s' not found in the load balancer.\n", task);*  *}*  *int main() {*  *struct LoadBalancer lb;*  *initializeLoadBalancer(&lb);*  *// Insert some tasks into the load balancer*  *insertTask(&lb, "Task1");*  *insertTask(&lb, "Task2");*  *insertTask(&lb, "Task3");*  *insertTask(&lb, "Task4");*  *insertTask(&lb, "Task5");*  *insertTask(&lb, "Task6");*  *// Remove a task from the load balancer*  *removeTask(&lb, "Task3");*  *// Try to remove a non-existing task*  *removeTask(&lb, "Task7");*  *return 0;*  *}* |
| **Sample Input:** |
| *Load Balancer Size: 10*  *Insertions (Some tasks to add to the load balancer):*  *- Task1*  *- Task2*  *- Task3*  *- Task4*  *- Task5*  *- Task6*  *Removals:*  *- Task3*  *- Task7* |

|  |
| --- |
| **Sample Output:** |
| *Task 'Task1' added to the load balancer.*  *Task 'Task2' added to the load balancer.*  *Task 'Task3' added to the load balancer.*  *Task 'Task4' added to the load balancer.*  *Task 'Task5' added to the load balancer.*  *Task 'Task6' added to the load balancer.*  *Task 'Task3' removed from the load balancer.*  *Task 'Task7' not found in the load balancer.* |