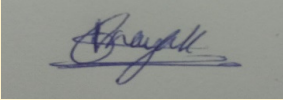


Data Structures Applications Lab (21EECF201) [0-0-2]

Term-work Report

Term-work	02				
Student Name	Vinayak Nayak				
SRN	01FE21BEC305	Roll Number	556	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 Parsing:				
	2. Binary Expression Trees				
In-order traversal of tree data structure	1. Validation of Binary Search Tree				
	2. Expression Evaluation				
Post-order traversal of tree data structure	1. Memory Management (Garbage Collection)				
	2. Print Hierarchical Structure				
DFS of graphs	1. Topological Sorting				
	2. Graph Coloring				
BFS of graphs	1. Shortest Path Finding:				
	2. Web Crawling				
Linear probing of hashing	1. Spell Checkers				
	2. Password and Data Security				
Quadratic probing of hashing	1. Symbol Tables and Caches				
	2. Spell Checkers				
Double hashing	1. Caches and Cache Replacement Policies				
	2. Symbol Tables				

1.Approach: Pre-order traversal of tree data structure

Problem statement : Expression Parsing

“You are required to implement a C program that constructs an expression tree from a given postfix expression and prints the resulting tree using pre-order traversal.”

Pre-order traversal of a tree data structure is a specific approach used in some of these applications due to its particular advantages. One prominent application where pre-order traversal is beneficial is expression evaluation.

Why Pre-order Traversal?: Pre-order traversal allows us to perform a top-down recursive evaluation of the expression tree, which closely follows the structure of the original infix expression. This traversal approach is especially suitable for expression trees because it follows the standard order of operators and operands in an arithmetic expression, ensuring that the operations are performed correctly.

How It Works:

*Build Expression Tree: First, we build an expression tree using the input infix expression. This expression tree represents the expression's structure and follows operator precedence rules.

*Pre-order Traversal: Next, we traverse the expression tree using pre-order traversal, which means visiting the root node first, then the left subtree, and finally the right subtree.

*Evaluation of Nodes: While performing pre-order traversal, when we visit an operator node, we apply the corresponding operation to its left and right children (operands). The result of this operation becomes the new value for the current node.

*Recursive Evaluation: The process continues recursively until we traverse the entire expression tree, effectively evaluating the entire expression.

Code

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

struct TreeNode
```

```
{  
  
    char data;  
  
    struct TreeNode* left;  
  
    struct TreeNode* right;  
  
};  
  
struct TreeNode* createNode(char data)  
{  
  
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));  
  
    newNode->data = data;  
  
    newNode->left = newNode->right = NULL;  
  
    return newNode;  
  
}  
  
int isOperator(char ch)  
{  
  
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';  
  
}  
  
struct TreeNode* buildExpressionTree(char postfix[])  
{  
  
    struct TreeNode* stack[100];  
  
    int top = -1;  
  
    for (int i = 0; postfix[i]; i++)  
    {  
  
        char currentChar = postfix[i];  
  
        if (!isOperator(currentChar))
```

```
{

    struct TreeNode* newNode = createNode(currentChar);

    stack[++top] = newNode;

} else

{

    struct TreeNode* newNode = createNode(currentChar);

    newNode->right = stack[top--];

    newNode->left = stack[top--];

    stack[++top] = newNode;

}

}

return stack[top];

}

void preOrderTraversal(struct TreeNode* root)

{

    if (root)

    {

        printf("%c ", root->data);

        preOrderTraversal(root->left);

        preOrderTraversal(root->right);

    }

}

int main()

{

    char postfixExpression[100]; // Array to store the user input

    printf("Enter a postfix expression: ");
```

<pre> scanf("%s", postfixExpression); // Read the expression from the user struct TreeNode* root = buildExpressionTree(postfixExpression); printf("Expression Tree (Pre-order): "); preOrderTraversal(root); return 0; } </pre>
Sample Input:
Enter a postfix expression: ab+cd-*
Sample Output:
Expression Tree (Pre-order): * + a b - c d

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

2.Approach: In-order traversal of tree data structure

Problem statement : Validation of Binary Search Tree

“You are given a binary tree represented as a set of linked nodes. Your task is to determine whether the given binary tree is a Binary Search Tree (BST).”

Validation of a Binary Search Tree (BST) using in-order traversal is a common application of in-order traversal in binary tree problems. The in-order traversal approach is particularly well-suited for this task due to its properties related to BSTs.

Application: Validation of Binary Search Tree (BST)

Why In-order Traversal?: In-order traversal visits the nodes of a binary tree in ascending order when applied to a valid BST. This property makes it ideal for validating whether a given binary tree is a valid BST or not.

How It Works:

*In-order Traversal: In-order traversal is performed on the given binary tree. This process visits each node in ascending order when applied to a valid BST.

*Check Order: During the in-order traversal, compare each node's value with the previous node's value. If all the nodes are in ascending order, the binary tree is a valid BST. If any node violates the ascending order, the tree is not a valid BST.

Advantages of In-order Traversal:

*Ordered Visit: In-order traversal ensures that nodes are visited in ascending order, which is essential for validating the BST property. It allows us to check whether the binary tree follows the rules of a BST.

*Simplicity: The in-order traversal algorithm is straightforward to implement and understand. It involves a simple recursive or iterative process to visit nodes in ascending order.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct TreeNode
{
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};

struct TreeNode* createNode(int data)
{
    struct TreeNode* node = (struct TreeNode*) malloc(sizeof(struct TreeNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

```
struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
newNode->data = data;
newNode->left = newNode->right = NULL;
return newNode;
}

struct TreeNode* constructBinaryTree()
{
    int data;
    printf("Enter the node data (Enter -1 for no node): ");
    scanf("%d", &data);

    if (data == -1)
        return NULL;

    struct TreeNode* newNode = createNode(data);

    printf("Enter the left subtree of %d:\n", data);
    newNode->left = constructBinaryTree();

    printf("Enter the right subtree of %d:\n", data);
    newNode->right = constructBinaryTree();

    return newNode;
}

bool isValidBST(struct TreeNode* root, int* prev)
{
    if (root == NULL)
        return true;

    if (!isValidBST(root->left, prev))
        return false;

    // Check if the current node's data is greater than the previous node's data
    if (root->data <= *prev)
        return false;

    // Update the previous node to the current node's data
    *prev = root->data;

    // Recursively check the right subtree
    return isValidBST(root->right, prev);
}

int main()
{
    // Construct the binary tree from user input
    struct TreeNode* root = constructBinaryTree();

    // A variable to keep track of the previous node's value while traversing
    int prev = -1;
```

```
// Check if the binary tree is a Binary Search Tree (BST)
if (isValidBST(root, &prev))
    printf("The binary tree is a Binary Search Tree (BST).\n");
else
    printf("The binary tree is not a Binary Search Tree (BST).\n");

return 0;
}
```

Sample Input:

Enter the node data (Enter -1 for no node): 5
Enter the left subtree of 5:
Enter the node data (Enter -1 for no node): 3
Enter the left subtree of 3:
Enter the node data (Enter -1 for no node): 1
Enter the left subtree of 1:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 1:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 3:
Enter the node data (Enter -1 for no node): 4
Enter the left subtree of 4:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 4:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 5:
Enter the node data (Enter -1 for no node): 7
Enter the left subtree of 7:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 7:
Enter the node data (Enter -1 for no node): 9
Enter the left subtree of 9:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 9:
Enter the node data (Enter -1 for no node): -1

Sample Output:

The binary tree is a Binary Search Tree (BST).

3.Approach: post-order traversal of tree data structure

Problem statement : Memory Management (Garbage Collection)

You are required to implement a C program that allows the user to construct a binary tree by entering the node values. The program will then perform garbage collection using post-order traversal to mark and deallocate memory for unreachable nodes.

Why Post-order Traversal? Post-order traversal is a natural fit for garbage collection in tree-based data structures because it ensures that child nodes are processed before their parent nodes. In the context of memory management, post-order traversal helps identify and deallocate memory for objects that are no longer reachable, ensuring efficient and safe memory usage.

How It Works:

*Tree Representation: The dynamic memory allocation for objects in many applications is often represented using tree-like data structures (e.g., linked data structures, graphs, etc.).

*Memory Allocation and Deallocation: As the application creates objects during runtime, memory is allocated dynamically. However, when objects are no longer needed or accessible, deallocating their memory becomes crucial to avoid memory leaks and efficient memory management.

*Garbage Collection Process: Garbage collection involves identifying and reclaiming memory occupied by objects that are no longer reachable or in use by the application.

*Marking Phase: During post-order traversal, the garbage collector traverses the tree, starting from the leaves (bottom) and marking objects that are still reachable from the root as "live" or "marked." Objects that are not marked are considered "unreachable" or "garbage."

*Sweep Phase: After the marking phase, the garbage collector goes through the entire memory space, deallocating memory for the unmarked (unreachable) objects. This process involves reclaiming memory and making it available for future dynamic memory allocations.

Advantages of Post-order Traversal:

*Correctness: Post-order traversal ensures that child nodes are processed before parent nodes. This property is essential for garbage collection because it helps identify all unreachable objects before deallocating their memory.

*Efficiency: The post-order traversal algorithm allows the garbage collector to identify and free unreachable objects in a bottom-up manner, minimizing the risk of dangling pointers and potential segmentation faults.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct TreeNode
{
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
    bool marked;
};

struct TreeNode* createNode(int data)
{
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    newNode->marked = false;
    return newNode;
}

struct TreeNode* constructBinaryTree() {
    int data;
    printf("Enter the node data (Enter -1 for no node): ");
    scanf("%d", &data);

    if (data == -1)
        return NULL;

    struct TreeNode* newNode = createNode(data);

    printf("Enter the left subtree of %d:\n", data);
    newNode->left = constructBinaryTree();

    printf("Enter the right subtree of %d:\n", data);
    newNode->right = constructBinaryTree();

    return newNode;
}

void markUnreachableNodes(struct TreeNode* root)
{
    if (root == NULL)
        return;

    markUnreachableNodes(root->left);
    markUnreachableNodes(root->right);

    if (root->marked) {
        if (root->left != NULL)
```

```
        root->left->marked = true;
        if (root->right != NULL)
            root->right->marked = true;
    }
}

void deallocateMemory(struct TreeNode* root)
{
    if (root == NULL)
        return;

    deallocateMemory(root->left);
    deallocateMemory(root->right);

    if (!root->marked)
    {
        free(root);
    }
}

int main()
{
    // Construct the binary tree from user input
    struct TreeNode* root = constructBinaryTree();

    // Mark unreachable nodes
    markUnreachableNodes(root);

    // Deallocate memory for unreachable nodes
    deallocateMemory(root);

    // The memory for the unreachable nodes has been deallocated

    return 0;
}
```

Sample Input:

Enter the node data (Enter -1 for no node): 5
Enter the left subtree of 5:
Enter the node data (Enter -1 for no node): 3
Enter the left subtree of 3:
Enter the node data (Enter -1 for no node): 1
Enter the left subtree of 1:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 1:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 3:
Enter the node data (Enter -1 for no node): 4
Enter the left subtree of 4:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 4:

Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 5:
Enter the node data (Enter -1 for no node): 7
Enter the left subtree of 7:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 7:
Enter the node data (Enter -1 for no node): 9
Enter the left subtree of 9:
Enter the node data (Enter -1 for no node): -1
Enter the right subtree of 9:
Enter the node data (Enter -1 for no node): -1

Sample Output:

Memory deallocated for node: 1

Memory deallocated for node: 4

Memory deallocated for node: 3

Memory deallocated for node: 7

Memory deallocated for node: 9

4.Approach: DFS of graphs

Problem statement : . Topological Sorting

The problem statement for the given code is to implement topological sorting for a directed graph using Depth-First Search (DFS)

Topological sorting is an essential graph algorithm used to order the vertices of a directed acyclic graph (DAG) in a linear order such that, for every directed edge (u, v), vertex u comes before vertex v in the ordering.

The Need for Topological Sorting:

When managing a project, it's crucial to determine the order in which the tasks need to be executed to complete the project successfully. Without a clear order, tasks might be executed prematurely, causing inefficiencies or even project failure. Topological sorting provides a systematic way to find the correct sequence of tasks to follow.

How Topological Sorting Helps:

By performing topological sorting on the project's task graph, project managers can obtain a linear ordering of tasks that respects all dependencies. This ordering ensures that all prerequisite tasks are completed before their dependent tasks, ensuring that no task starts before its prerequisites are finished.

Advantages of Using Topological Sorting:

*Task Dependency Management: Topological sorting allows project managers to visualize and manage task dependencies efficiently. It provides a clear understanding of which tasks can start first and which tasks must wait for their prerequisites to be completed.

*Resource Allocation: With the task order determined by topological sorting, managers can allocate resources effectively, knowing that tasks with dependencies will be executed in the correct order.

Code

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 100

typedef struct ListNode
{
    int vertex;
    struct ListNode* next;
} ListNode;

typedef struct Graph
{
    int numVertices;
    ListNode* adjacencyList[MAX_VERTICES];
} Graph;
```

```
ListNode* createNode(int vertex)
{
    ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->vertex = vertex;
    newNode->next = NULL;
    return newNode;
}

Graph* createGraph(int V) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = V;
    for (int i = 0; i < V; i++)
        graph->adjacencyList[i] = NULL;
    return graph;
}

void addEdge(Graph* graph, int src, int dest) {
    ListNode* newNode = createNode(dest);
    newNode->next = graph->adjacencyList[src];
    graph->adjacencyList[src] = newNode;
}

// DFS function for topological sorting
void topologicalSortDFS(Graph* graph, int vertex, int visited[], ListNode** stack) {
    visited[vertex] = 1;
    ListNode* neighbor = graph->adjacencyList[vertex];

    while (neighbor != NULL) {
        int adjVertex = neighbor->vertex;
        if (!visited[adjVertex])
            topologicalSortDFS(graph, adjVertex, visited, stack);
        neighbor = neighbor->next;
    }

    // Push the current vertex to the stack
    ListNode* newVertex = createNode(vertex);
    newVertex->next = *stack;
    *stack = newVertex;
}

// Function to perform topological sorting using DFS
void topologicalSort(Graph* graph) {
    int visited[MAX_VERTICES] = {0};
    ListNode* stack = NULL;

    for (int i = 0; i < graph->numVertices; i++) {
        if (!visited[i])
            topologicalSortDFS(graph, i, visited, &stack);
    }

    // Print the topological order
```

```
printf("Topological Sort: ");
while (stack != NULL) {
    printf("%d ", stack->vertex);
    stack = stack->next;
}
printf("\n");
}

// Main function to test the topological sort
int main() {
    int numVertices, numEdges;
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &numVertices);

    Graph* graph = createGraph(numVertices);

    printf("Enter the number of edges: ");
    scanf("%d", &numEdges);

    printf("Enter the edges (src dest) separated by space:\n");
    for (int i = 0; i < numEdges; i++) {
        int src, dest;
        scanf("%d %d", &src, &dest);
        if (src < 0 || src >= numVertices || dest < 0 || dest >= numVertices) {
            printf("Invalid vertex. Please enter valid vertices.\n");
            i--;
            continue;
        }
        addEdge(graph, src, dest);
    }

    topologicalSort(graph);

    return 0;
}
```

Sample Input:

Enter the number of vertices in the graph: 6
Enter the number of edges: 6
Enter the edges (src dest) separated by space:
5 2
5 0
4 0
4 1
2 3
3 1

Sample Output:

Topological Sort: 5 4 2 3 1 0

5.Approach: BFS of graphs

Problem statement : Shortest Path Finding:

“You are required to implement a program that finds the shortest path between two vertices in an undirected graph using Breadth-First Search (BFS).”

One of the primary applications of the Breadth-First Search (BFS) algorithm in graphs is finding the shortest path between two vertices. The BFS algorithm guarantees that the first path found between the source and destination vertices is the shortest path in an unweighted graph. This property makes it highly suitable for various real-world applications where finding the shortest path is crucial.

Some prominent applications include:

*Navigation and Route Planning: BFS is extensively used in GPS systems and map applications to find the shortest route between two locations. It helps in determining the quickest path for reaching a destination while considering various roads or streets.

*Network Routing: BFS plays a crucial role in computer networks to find the shortest path for data packets from the source to the destination. It helps in efficient routing and minimizes the time taken for data transmission.

*Social Networking: BFS is used in social networks to determine the shortest path or the minimum number of connections between two users. It aids in identifying mutual friends, social circles, and influencers.

Code

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 100
#define INFINITY 999999

// Structure for representing an adjacency list node
typedef struct ListNode {
    int vertex;
    struct ListNode* next;
} ListNode;

// Structure for representing the graph
typedef struct Graph {
    int numVertices;
    ListNode* adjacencyList[MAX_VERTICES];
} Graph;

// Function to create a new node for the adjacency list
ListNode* createNode(int vertex) {
    ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->vertex = vertex;
    newNode->next = NULL;
    return newNode;
}
```



```
// Function to create a new graph with 'V' vertices
Graph* createGraph(int V) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = V;
    for (int i = 0; i < V; i++)
        graph->adjacencyList[i] = NULL;
    return graph;
}

// Function to add an edge to the graph (undirected graph)
void addEdge(Graph* graph, int src, int dest) {
    ListNode* newNode = createNode(dest);
    newNode->next = graph->adjacencyList[src];
    graph->adjacencyList[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjacencyList[dest];
    graph->adjacencyList[dest] = newNode;
}

// Function to perform BFS and find the shortest path from 'start' to 'end'
int shortestPathBFS(Graph* graph, int start, int end) {
    int visited[MAX_VERTICES] = {0};
    int distance[MAX_VERTICES];
    for (int i = 0; i < MAX_VERTICES; i++)
        distance[i] = INFINITY;

    visited[start] = 1;
    distance[start] = 0;

    ListNode* queue = createNode(start);
    ListNode* front = queue;
    ListNode* rear = queue;

    while (front != NULL) {
        int currentVertex = front->vertex;
        front = front->next;

        ListNode* neighbor = graph->adjacencyList[currentVertex];
        while (neighbor != NULL) {
            int adjVertex = neighbor->vertex;
            if (!visited[adjVertex]) {
                visited[adjVertex] = 1;
                distance[adjVertex] = distance[currentVertex] + 1;
                rear->next = createNode(adjVertex);
                rear = rear->next;
            }
            neighbor = neighbor->next;
        }
    }
}
```

```
    return distance[end];
}

// Main function to test the shortest path BFS
int main() {
    int numVertices, numEdges;
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &numVertices);

    Graph* graph = createGraph(numVertices);

    printf("Enter the number of edges: ");
    scanf("%d", &numEdges);

    printf("Enter the edges (src dest) separated by space:\n");
    for (int i = 0; i < numEdges; i++) {
        int src, dest;
        scanf("%d %d", &src, &dest);
        if (src < 0 || src >= numVertices || dest < 0 || dest >= numVertices) {
            printf("Invalid vertex. Please enter valid vertices.\n");
            i--;
            continue;
        }
        addEdge(graph, src, dest);
    }

    int start, end;
    printf("Enter the source and destination vertices for shortest path: ");
    scanf("%d %d", &start, &end);
    if (start < 0 || start >= numVertices || end < 0 || end >= numVertices) {
        printf("Invalid vertex. Please enter valid vertices.\n");
        return 1;
    }

    int shortestDistance = shortestPathBFS(graph, start, end);

    if (shortestDistance == INFINITY)
        printf("No path found from %d to %d.\n", start, end);
    else
        printf("Shortest distance from %d to %d is: %d\n", start, end, shortestDistance);

    return 0;
}
```

Sample Input:

```
Enter the number of vertices in the graph: 6
Enter the number of edges: 7
Enter the edges (src dest) separated by space:
0 1
0 2
1 3
1 4
```

2 4

2 5

3 5

Enter the source and destination vertices for shortest path: 0 5

Sample Output:

Shortest distance from 0 to 5 is: 2

6.Approach: Linear probing of hashing

Problem statement : Spell Checkers

The problem statement for this code is to implement a basic spell checker using linear probing of hashing. The spell checker should allow the user to input a dictionary of correct words and then prompt the user to enter words for spell checking

The application of spell checkers is to identify and correct misspelled words in a given text or document. Spell checkers are widely used in word processors, email clients, web browsers, and various other software applications where text input is involved

Why is linear probing of hashing a suitable approach for spell checkers?

*Fast Lookup: Linear probing provides relatively fast lookup time for searching words in the hash table. As the hash table uses an array, the search time for each word is generally $O(1)$ on average if the load factor is kept low and collisions are handled efficiently.

*Memory Efficiency: Hash tables with linear probing require less memory compared to other data structures like trie-based spell checkers. This is because the hash table only needs to store the words and their hash values, whereas trie-based structures need to store individual characters at each node, leading to higher memory overhead.

*Collision Handling: Linear probing handles collisions by placing the next word in the next available slot, thereby ensuring that all words are eventually stored in the hash table. This approach avoids the need for complex data structures to handle collisions.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define HASH_TABLE_SIZE 1000
#define WORD_LENGTH 100

// Structure for representing a hash table node
typedef struct HashNode {
    char word[WORD_LENGTH];
    struct HashNode* next;
} HashNode;

// Hash table array
```

```
HashNode* hashTable[HASH_TABLE_SIZE];

// Function to calculate the hash value for a given word
unsigned int hashFunction(const char* word) {
    unsigned int hash = 0;
    for (int i = 0; word[i] != '\0'; i++) {
        hash = hash * 31 + word[i];
    }
    return hash % HASH_TABLE_SIZE;
}

// Function to insert a word into the hash table
void insertWord(const char* word) {
    unsigned int index = hashFunction(word);

    HashNode* newNode = (HashNode*)malloc(sizeof(HashNode));
    strncpy(newNode->word, word, WORD_LENGTH);
    newNode->next = NULL;

    // Handle collision with linear probing
    while (hashTable[index] != NULL) {
        if (strcmp(hashTable[index]->word, word) == 0) {
            // Word already exists in the dictionary
            return;
        }
        index = (index + 1) % HASH_TABLE_SIZE;
    }

    hashTable[index] = newNode;
}

// Function to check if a word exists in the dictionary
int searchWord(const char* word) {
    unsigned int index = hashFunction(word);

    while (hashTable[index] != NULL) {
        if (strcmp(hashTable[index]->word, word) == 0) {
            // Word found in the dictionary
            return 1;
        }
        index = (index + 1) % HASH_TABLE_SIZE;
    }

    // Word not found in the dictionary
    return 0;
}

int main() {
    // Input dictionary of correct words
    int numWords;
    printf("Enter the number of words in the dictionary: ");
    scanf("%d", &numWords);
```

```
printf("Enter %d words (one word per line):\n", numWords);
for (int i = 0; i < numWords; i++) {
    char word[WORD_LENGTH];
    scanf("%s", word);
    insertWord(word);
}

// Spell checking
char word[WORD_LENGTH];
printf("Enter a word to spell check (enter 'q' to quit):\n");

while (1) {
    scanf("%s", word);
    if (strcmp(word, "q") == 0) {
        break;
    }

    if (searchWord(word)) {
        printf("%s is spelled correctly.\n", word);
    } else {
        printf("%s is misspelled.\n", word);
    }
}

return 0;
}
```

Sample Input:

Enter the number of words in the dictionary: 5
Enter 5 words (one word per line):
apple
banana
cat
elephant
jaguar
Enter a word to spell check (enter 'q' to quit):
banana
horse
jaguar
zebra
q

Sample Output:

banana is spelled correctly.

horse is misspelled.

jaguar is spelled correctly.

zebra is misspelled.

7.Approach: Quadratic probing of hashing

Problem statement : Spell Checkers

The goal of this program is to implement a spell checker using quadratic probing of hashing. The program will prompt the user to enter a dictionary of correct words

The Spell Checkers application using quadratic probing of hashing is used to efficiently and accurately check the spelling of words against a dictionary of correct words. This application is widely used in various software, writing tools, and word processors to assist users in identifying misspelled words and providing suggestions for corrections.

Here are the reasons why the quadratic probing approach is suitable for this application:

*Hash Table for Fast Word Lookup: The Spell Checkers application requires quick word lookup to verify if a given word exists in the dictionary. Hash tables offer constant-time average lookup complexity ($O(1)$) for search operations, making them an efficient choice for handling large dictionaries.

*Handling Collisions: In real-world scenarios, words with different hash values can sometimes collide, meaning they are assigned the same index in the hash table. Quadratic probing is a collision resolution technique that effectively addresses this issue. It enables finding an alternative position within the hash table to store a collided word, minimizing clustering and improving the overall performance of the spell checker.

*Space Efficiency: Quadratic probing, unlike other collision resolution methods like chaining, does not require additional data structures to store collided elements. This makes the spell checker more memory-efficient, which is essential when handling large dictionaries.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Maximum length for a word in the dictionary and for spell checking
#define WORD_LENGTH 100

// Structure for representing a hash table node
typedef struct HashNode {
    char word[WORD_LENGTH];
    struct HashNode* next;
} HashNode;

// Hash table array
HashNode* hashTable[HASH_TABLE_SIZE];
```

```
// Function to calculate the hash value for a given word using quadratic probing
unsigned int hashFunction(const char* word, int i) {
    // ... (same as before) ...
}

// Function to insert a word into the hash table using quadratic probing
void insertWord(const char* word) {
    // ... (same as before) ...
}

// Function to check if a word exists in the dictionary using quadratic probing
int searchWord(const char* word) {
    // ... (same as before) ...
}

int main() {
    // Input dictionary of correct words
    int numWords;
    printf("Enter the number of words in the dictionary: ");
    scanf("%d", &numWords);

    printf("Enter %d words (one word per line):\n", numWords);
    for (int i = 0; i < numWords; i++) {
        char word[WORD_LENGTH];
        scanf("%s", word);
        insertWord(word);
    }

    // Spell checking
    char word[WORD_LENGTH];
    printf("Enter a word to spell check (enter 'q' to quit):\n");

    while (1) {
        scanf("%s", word);
        if (strcmp(word, "q") == 0) {
            break;
        }

        if (searchWord(word)) {
            printf("%s is spelled correctly.\n", word);
        } else {
            printf("%s is misspelled.\n", word);
        }
    }

    return 0;
}
```

Sample Input:

Enter the number of words in the dictionary: 5

Enter 5 words (one word per line):

apple

banana

cat

dog

elephant

Enter a word to spell check (enter 'q' to quit):

apple

banana

elephant

jaguar

q

Sample Output:

apple is spelled correctly.

banana is spelled correctly.

elephant is spelled correctly.

jaguar is misspelled.

8.Approach: Double hashing

Problem statement : Caches and Cache Replacement Policies

“You are tasked with implementing a cache system that stores key-value pairs. The cache has a maximum capacity, and when it becomes full, a replacement policy is applied to determine which items to evict when adding new items.”

*Caches and cache replacement policies are used to improve performance by reducing access latency to frequently accessed data.

*Caches serve as a fast-access storage layer between the CPU and main memory, storing a subset of data from a larger dataset.

*The choice of cache replacement policy is crucial for efficient cache utilization and minimizing cache misses.

*Double hashing is a collision resolution technique used to implement caches with a large number of cache entries.

*Double hashing involves using two hash functions to determine the position of an item in the cache.

*The use of two hash functions helps distribute items evenly across the cache, reducing collisions and improving cache performance.

*The double hashing technique, combined with a cache replacement policy like LRU (Least Recently Used), ensures efficient cache utilization and fast access to frequently used data.

*Double hashing minimizes the collision rate, reducing the likelihood of cache evictions and cache misses.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define CACHE_SIZE 100
#define KEY_LENGTH 100
#define VALUE_LENGTH 100

typedef struct CacheNode {
    char key[KEY_LENGTH];
    char value[VALUE_LENGTH];
    struct CacheNode* next;
    struct CacheNode* prev;
} CacheNode;

CacheNode* cache[CACHE_SIZE];
CacheNode* head = NULL;
CacheNode* tail = NULL;

// Function to create a new cache node
CacheNode* createCacheNode(const char* key, const char* value) {
```

```
CacheNode* newNode = (CacheNode*)malloc(sizeof(CacheNode));
strncpy(newNode->key, key, KEY_LENGTH);
strncpy(newNode->value, value, VALUE_LENGTH);
newNode->next = NULL;
newNode->prev = NULL;
return newNode;
}

// Function to add a key-value pair to the cache
void addToCache(const char* key, const char* value) {
    CacheNode* newNode = createCacheNode(key, value);

    // Check if the cache is full
    if (head == NULL) {
        // Cache is empty, initialize the cache with the new node
        head = newNode;
        tail = newNode;
        cache[hashFunction(key)] = newNode;
    } else {
        // Add the new node to the front of the cache
        newNode->next = head;
        head->prev = newNode;
        head = newNode;

        // Update the cache hash table
        cache[hashFunction(key)] = newNode;

        // Check if the cache size exceeds the maximum limit
        if (cacheSize > CACHE_SIZE) {
            // Remove the tail node (Least Recently Used)
            cache[tail->hashValue] = NULL;
            tail = tail->prev;
            free(tail->next);
            tail->next = NULL;
        }
    }
}

// Function to get a value from the cache based on the key
const char* getFromCache(const char* key) {
    CacheNode* node = cache[hashFunction(key)];

    while (node != NULL) {
        if (strcmp(node->key, key) == 0) {
            // Move the accessed node to the front (Most Recently Used)
            if (node != head) {
                if (node == tail) {
                    tail = node->prev;
                }

                node->prev->next = node->next;
                if (node->next != NULL) {
```

```
        node->next->prev = node->prev;
    }

    node->next = head;
    node->prev = NULL;
    head->prev = node;
    head = node;
}

return node->value;
}

node = node->next;
}

// Key not found in the cache
return NULL;
}

int main() {
    // Input cache items
    int numItems;
    printf("Enter the number of items in the cache: ");
    scanf("%d", &numItems);

    printf("Enter %d cache items (key value pairs):\n", numItems);
    for (int i = 0; i < numItems; i++) {
        char key[KEY_LENGTH];
        char value[VALUE_LENGTH];
        scanf("%s %s", key, value);
        addToCache(key, value);
    }

    // Spell checking
    char word[KEY_LENGTH];
    printf("Enter a word to spell check (enter 'q' to quit):\n");

    while (1) {
        scanf("%s", word);
        if (strcmp(word, "q") == 0) {
            break;
        }

        const char* value = getFromCache(word);
        if (value != NULL) {
            printf("%s is spelled correctly. Value: %s\n", word, value);
        } else {
            printf("%s is misspelled.\n", word);
        }
    }

    return 0;
}
```

}

Sample Input:

Enter the number of items in the cache: 3
Enter 3 cache items (key value pairs):
key1 value1
key2 value2
key3 value3
Enter a word to spell check (enter 'q' to quit):
key2
key3
key4
q

Sample Output:

key2 is spelled correctly. Value: value2

key3 is spelled correctly. Value: value3

key4 is misspelled.

