seperate chaining

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

#define TABLE_SIZE 7  // Size of the hash table (fixed size)

// Node structure representing a record in the hash table
typedef struct node {
    int rollNo;           // Unique identifier (key) for the student
    char studentName[25]; // Name of the student (value)
    struct node *next;    // Pointer to the next node in case of a collision
(chaining)
} Node;

// Function prototypes
void initializeTable(Node *hashTable[]);               // Initialize the
hash table
int hashFunction(int key);                             // Hash function to
map roll numbers to table indices
void insertRecord(Node *hashTable[], int rollNo, char studentName[]); //
Insert a new record
int deleteRecord(Node *hashTable[], int rollNo);       // Delete a record by
roll number
int searchRecord(Node *hashTable[], int rollNo, char studentName[]);  //
Search for a record
void displayTable(Node *hashTable[]);                  // Display all
records in the hash table
void destroyTable(Node *hashTable[]);                  // Free all allocated
memory and clear the table

int main() {
    Node *hashTable[TABLE_SIZE]; // Hash table implemented as an array of
Node pointers
    initializeTable(hashTable); // Initialize the hash table to NULL
    int choice, rollNo;
    char studentName[25];      // Temporary storage for user input
```

```c
    do {
        // Display the menu options to the user
        printf("1. Insert\n2. Delete\n3. Search\n4. Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Insert a new record into the hash table
                printf("Enter the roll number: ");
                scanf("%d", &rollNo);
                fflush(stdin); // Clear input buffer to handle string input
correctly
                printf("Enter the name: ");
                scanf("%s[^\n]", studentName); // Read full name (including
spaces)
                insertRecord(hashTable, rollNo, studentName);
                break;
            case 2: // Delete a record by roll number
                printf("Enter the roll number: ");
                scanf("%d", &rollNo);
                if (deleteRecord(hashTable, rollNo) == 0) // If deletion
fails
                    printf("Record not found\n");
                break;
            case 3: // Search for a record by roll number
                printf("Enter the roll number: ");
                scanf("%d", &rollNo);
                if (searchRecord(hashTable, rollNo, studentName)) // If
found, display the name
                    printf("Record found, name = %s\n", studentName);
                else
                    printf("Record not found\n");
                break;
            case 4: // Display all records in the hash table
                displayTable(hashTable);
                break;
        }
    } while (choice < 5); // Exit the loop when the user chooses option 5
    destroyTable(hashTable); // Clean up allocated memory before exiting
    return 0;
}

// Initialize the hash table by setting all slots to NULL
void initializeTable(Node *hashTable[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = NULL; // Each index starts as an empty chain
    }
}

// Hash function to compute the index for a given roll number
int hashFunction(int key) {
    return key % TABLE_SIZE; // Use modulo operation for simplicity
}

// Insert a new record into the hash table
void insertRecord(Node *hashTable[], int rollNo, char studentName[]) {
    int index = hashFunction(rollNo); // Compute the hash index for the roll
number

    // Create a new node for the record
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->rollNo = rollNo; // Set the roll number
    strcpy(newNode->studentName, studentName); // Copy the name into the
node
    newNode->next = hashTable[index]; // Point to the current chain at the
index
    hashTable[index] = newNode; // Update the head of the chain to the new
node
}

// Delete a record from the hash table
int deleteRecord(Node *hashTable[], int rollNo) {
    int index = hashFunction(rollNo); // Compute the hash index for the roll
number

    Node *current = hashTable[index]; // Start at the head of the chain
    Node *previous = NULL;            // Keep track of the previous node
    // Traverse the chain to find the node with the matching roll number
    while (current != NULL && current->rollNo != rollNo) {
        previous = current; // Move previous to the current node
        current = current->next; // Move current to the next node
    }

    if (current != NULL) { // If the node is found
        if (previous != NULL) { // If it's not the first node in the chain
            previous->next = current->next; // Remove the node by linking
around it
        } else {
            hashTable[index] = current->next; // Update the head of the
chain
        }
        free(current); // Free the memory for the deleted node
        return 1;      // Return success
    }
    return 0; // Node not found, return failure
}

// Search for a record in the hash table
int searchRecord(Node *hashTable[], int rollNo, char studentName[]) {
    int index = hashFunction(rollNo); // Compute the hash index for the roll
number

    Node *current = hashTable[index]; // Start at the head of the chain

    // Traverse the chain to find the node with the matching roll number
    while (current != NULL) {
        if (current->rollNo == rollNo) { // If found
            strcpy(studentName, current->studentName); // Copy the name into
the output variable
            return 1; // Return success
        }
        current = current->next; // Move to the next node
    }
    return 0; // Node not found, return failure
}

// Display the contents of the hash table
```

```
void displayTable(Node *hashTable[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("hashTable[%d] -> ", i); // Display the index

        Node *current = hashTable[i]; // Start at the head of the chain
        while (current != NULL) { // Traverse the chain
            printf("%d %s -> ", current->rollNo, current->studentName); //
Print each record
            current = current->next; // Move to the next node
        }
        printf("NULL\n"); // Mark the end of the chain
    }
}

// Free all allocated memory in the hash table
void destroyTable(Node *hashTable[]) {
    for (int i = 0; i < TABLE_SIZE; i++) { // Iterate through each index
        Node *current = hashTable[i]; // Start at the head of the chain

        while (current != NULL) { // Traverse the chain
            Node *temp = current; // Store the current node
            current = current->next; // Move to the next node
            free(temp); // Free the current node
        }
        hashTable[i] = NULL; // Set the head of the chain to NULL
    }
}
```

linear probing

```
#include <stdio.h>
#include <string.h>
#define TABLE_SIZE 7  // Size of the hash table

// Node structure representing a record in the hash table
typedef struct node {
    int rollNo;       // Roll number (key)
    char studentName[25]; // Name of the student (value)
    int status;       // Status of the slot: -1 (empty), 1 (occupied)
```

```
} Node;

// Function prototypes
void initializeTable(Node hashTable[], int *numRecords);
int hashFunction(int key);
int insertRecord(Node hashTable[], int rollNo, char studentName[], int
*numRecords);
int deleteRecord(Node hashTable[], int rollNo, int *numRecords);
int searchRecord(Node hashTable[], int rollNo, char studentName[], int
numRecords);
void displayTable(Node hashTable[]);

int main() {
    Node hashTable[TABLE_SIZE]; // Hash table implemented as an array of
Node structures
    int numRecords;            // Counter for the number of occupied slots

    initializeTable(hashTable, &numRecords); // Initialize the hash table
    int choice, rollNo;
    char studentName[25];

    do {
        // Display menu options
        printf("1. Insert\n2. Delete\n3. Search\n4. Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Insert a new record
                printf("Enter roll number: ");
                scanf("%d", &rollNo);
                fflush(stdin); // Clear input buffer
                printf("Enter name: ");
                scanf("%s[^\n]", studentName); // Read full name (with
spaces)
                if (insertRecord(hashTable, rollNo, studentName,
&numRecords) == 0)
                    printf("Table is already full. Cannot insert.\n");
                break;
            case 2: // Delete a record
                printf("Enter roll number: ");
                scanf("%d", &rollNo);
                if (deleteRecord(hashTable, rollNo, &numRecords) == 0)
                    printf("Record not found. Cannot delete.\n");
                break;

            case 3: // Search for a record
                printf("Enter roll number: ");
                scanf("%d", &rollNo);
                if (searchRecord(hashTable, rollNo, studentName,
numRecords))
                    printf("Record found. Name: %s\n", studentName);
                else
                    printf("Record not found.\n");
                break;

            case 4: // Display all records in the hash table
                displayTable(hashTable);
                break;
        }
    } while (choice < 5); // Exit the loop if choice is 5 (Exit)

    return 0;
}

// Initialize the hash table by marking all slots as empty
void initializeTable(Node hashTable[], int *numRecords) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i].status = -1; // Mark slot as empty
    }
    *numRecords = 0; // No records in the table initially
}

// Hash function to compute the index for a given roll number
int hashFunction(int key) {
    return key % TABLE_SIZE; // Simple modulo operation
}

// Insert a new record into the hash table using linear probing
```

```
int insertRecord(Node hashTable[], int rollNo, char studentName[], int
*numRecords) {
    if (*numRecords == TABLE_SIZE) { // Check if the table is full
        return 0; // Insertion failed
    }

    int index = hashFunction(rollNo); // Compute the initial hash index

    // Linear probing to find the next available slot
    while (hashTable[index].status == 1) { // While the slot is occupied
        index = (index + 1) % TABLE_SIZE; // Move to the next slot
(circularly)
    }

    // Insert the new record into the available slot
    hashTable[index].rollNo = rollNo;       // Store the roll number
    strcpy(hashTable[index].studentName, studentName); // Copy the student's
name
    hashTable[index].status = 1;           // Mark the slot as occupied
    (*numRecords)++;                       // Increment the count of records
    return 1; // Insertion successful
}

// Delete a record from the hash table
int deleteRecord(Node hashTable[], int rollNo, int *numRecords) {
    if (*numRecords == 0) { // Check if the table is empty
        return 0; // Deletion failed
    }

    int index = hashFunction(rollNo); // Compute the initial hash index
    int scannedSlots = 0;            // Track the number of slots scanned

    // Linear probing to find the record
    while (scannedSlots < *numRecords && hashTable[index].rollNo != rollNo)
{
        if (hashTable[index].status == 1) { // If the slot is occupied
            scannedSlots++;
        }
        index = (index + 1) % TABLE_SIZE; // Move to the next slot
(circularly)
```

```
    }

    // Check if the record was found
    if (hashTable[index].rollNo == rollNo && hashTable[index].status == 1) {
        hashTable[index].status = -1; // Mark the slot as deleted
        (*numRecords)--;             // Decrement the count of records
        return 1; // Deletion successful
    }
    return 0; // Record not found
}

// Search for a record in the hash table
int searchRecord(Node hashTable[], int rollNo, char studentName[], int
numRecords) {
    if (numRecords == 0) { // Check if the table is empty
        return 0; // Search failed
    }

    int index = hashFunction(rollNo); // Compute the initial hash index
    int scannedSlots = 0;            // Track the number of slots scanned

    // Linear probing to find the record
    while (scannedSlots < numRecords && hashTable[index].rollNo != rollNo) {
        if (hashTable[index].status == 1) { // If the slot is occupied
            scannedSlots++;
        }
        index = (index + 1) % TABLE_SIZE; // Move to the next slot
(circularly)
    }

    // Check if the record was found
    if (hashTable[index].rollNo == rollNo && hashTable[index].status == 1) {
        strcpy(studentName, hashTable[index].studentName); // Copy the
student's name
        return 1; // Search successful
    }
    return 0; // Record not found
}

// Display the contents of the hash table
```

```
void displayTable(Node hashTable[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i].status == 1) { // If the slot is occupied
            printf("hashTable[%d]: RollNo = %d, Name = %s\n", i,
hashTable[i].rollNo, hashTable[i].studentName);
        } else { // If the slot is empty or deleted
            printf("hashTable[%d]: Empty\n", i);
        }
    }
}
```

quadratic probing

```
#include <stdio.h>
#include <string.h>
#define SIZE 7

typedef struct node {
        int rNo;
        char name[25];
        int mark;
} NODE;

void initTable(NODE ht[], int *n);
int hashFunction(int key);
int insertRecord(NODE ht[], int rNo, char name[], int *n);
int deleteRecord(NODE ht[], int rNo, int *n);
int searchRecord(NODE ht[], int rNo, char name[], int n);
void displayTable(NODE ht[]);

int main() {
        NODE ht[SIZE];
        int n;

        initTable(ht, &n);
        int choice, rNo;
        char name[25];

        do {
```

```
		printf("1.Insert 2.Delete 3.Search 4.Display\n");
		scanf("%d", &choice);
		switch (choice) {
			case 1:
				printf("Enter roll no.\n");
				scanf("%d", &rNo);
				fflush(stdin);
				printf("Enter name\n");
				scanf("%[^\n]", name);
				if (insertRecord(ht, rNo, name, &n) == 0)
					printf("Table is already full or no 
suitable position found\n");
				break;
			case 2:
				printf("Enter roll no.\n");
				scanf("%d", &rNo);
				if (deleteRecord(ht, rNo, &n) == 0)
					printf("Record not found\n");
				break;
			case 3:
				printf("Enter roll no.\n");
				scanf("%d", &rNo);
				if (searchRecord(ht, rNo, name, n))
					printf("Record found, name is %s\n", 
name);

				else
					printf("Record not found\n");
				break;
			case 4:
				displayTable(ht);
				break;
		}
	} while (choice < 5);
}

void initTable(NODE ht[], int *n) {
	int i;
	for (i = 0; i < SIZE; i++)
		ht[i].mark = -1;
	(*n) = 0;
```

```
}

int hashFunction(int key) {
        return key % SIZE;
}

int insertRecord(NODE ht[], int rNo, char name[], int *n) {
        if (*n == SIZE)
                return 0;

        int index = hashFunction(rNo);
        int i = 0;

        // Quadratic probing loop
        while (ht[(index + i * i) % SIZE].mark != -1 && i < SIZE) {
                i++;
        }

        if (i == SIZE)
                return 0; // No empty slot found after probing

        int finalIndex = (index + i * i) % SIZE;
        ht[finalIndex].rNo = rNo;
        strcpy(ht[finalIndex].name, name);
        ht[finalIndex].mark = 1;
        (*n)++;

        return 1;
}

int deleteRecord(NODE ht[], int rNo, int *n) {
        if (*n == 0)
                return 0;

        int index = hashFunction(rNo);
        int i = 0;

        // Quadratic probing search for the record
        while (i < SIZE && (ht[(index + i * i) % SIZE].rNo != rNo ||
ht[(index + i * i) % SIZE].mark != 1)) {
```

```
                i++;
        }

        if (i == SIZE || ht[(index + i * i) % SIZE].mark != 1)
                return 0; // Record not found

        int finalIndex = (index + i * i) % SIZE;
        ht[finalIndex].mark = -1; // Mark as deleted
        (*n)--;

        return 1;
}

int searchRecord(NODE ht[], int rNo, char name[], int n) {
        if (n == 0)
                return 0;

        int index = hashFunction(rNo);
        int i = 0;

        // Quadratic probing search for the record
        while (i < SIZE && (ht[(index + i * i) % SIZE].rNo != rNo ||
ht[(index + i * i) % SIZE].mark != 1)) {
                i++;
        }

        if (i == SIZE || ht[(index + i * i) % SIZE].mark != 1)
                return 0; // Record not found

        int finalIndex = (index + i * i) % SIZE;
        strcpy(name, ht[finalIndex].name);

        return 1;
}

void displayTable(NODE ht[]) {
        int i;
        for (i = 0; i < SIZE; i++) {
                if (ht[i].mark == 1)
                        printf("ht[%d]: %d %s\n", i, ht[i].rNo, ht[i].name);
                else
                        printf("ht[%d]:\n", i);
        }
}
```

double hashing

```
#include <stdio.h>
#define SIZE 7 // Define the size of the hash table

// Structure for each node in the hash table
typedef struct node {
    int data; // Stores the key value
    int flag; // Flag to indicate the status of the slot: 0 = empty, 1 =
occupied
} NODE;

// Function to initialize the hash table
// Sets all slots in the hash table as empty (flag = 0)
void initializeHashTable(NODE *hashTable) {
    int i;
    for (i = 0; i < SIZE; i++) {
        hashTable[i].data = 0;   // Initialize the data as 0
        hashTable[i].flag = 0;   // Mark the slot as empty
    }
}

// Function to insert a key into the hash table using Double Hashing
// Double Hashing: hash = (h1(key) + i * h2(key)) % SIZE
// - h1(key) = key % SIZE
// - h2(key) = key % 5
void insert(int key, NODE *hashTable) {
    int hash, hash2, i = 0;

    // Compute the second hash value
    hash2 = key % 5; // Secondary hash function (should be independent of
SIZE)

    // Compute the initial hash index using h1(key) and h2(key)
    hash = (key % SIZE + i * hash2) % SIZE;
    // Probe for an empty slot using Double Hashing
    while (hashTable[hash].flag != 0 && i < SIZE) { // Check if the slot is
occupied
        i++; // Increment the probe number
        hash = (key % SIZE + i * hash2) % SIZE; // Update the hash index
    }

    // If an empty slot is found
    if (hashTable[hash].flag == 0) {
        hashTable[hash].data = key; // Insert the key into the slot
        hashTable[hash].flag = 1;   // Mark the slot as occupied
        printf("The data %d is inserted at %d\n", key, hash);
    } else { // If no empty slot is found after SIZE probes
        printf("\nData cannot be inserted\n");
    }
}

// Function to search for a key in the hash table using Double Hashing
void search(int key, NODE *hashTable) {
    int hash, hash2, i = 0;

    // Compute the secondary hash value
    hash2 = 7 - (key % 7); // Secondary hash function (independent of SIZE)

    // Compute the initial hash index
    hash = (key % SIZE + i * hash2) % SIZE;

    // Probe to find the key in the hash table
    while (i < SIZE) { // Limit the number of probes to the size of the hash
table
        if (hashTable[hash].data == key && hashTable[hash].flag == 1) {
            // Key is found in the hash table
            printf("\nThe data %d is found at location %d\n", key, hash);
            return;
        }
        i++; // Increment the probe number
        hash = (key % SIZE + i * hash2) % SIZE; // Update the hash index
    }
    // If the loop completes without finding the key
    printf("\nData not found\n");
}

// Function to display the contents of the hash table
void display(NODE *hashTable) {
    int i;
    printf("\nHash Table:\n");
    for (i = 0; i < SIZE; i++) {
        if (hashTable[i].flag == 1) { // If the slot is occupied
            printf("Index %d: %d\n", i, hashTable[i].data);
        } else { // If the slot is empty
            printf("Index %d: EMPTY\n", i);
        }
    }
    printf("\n");
}

int main() {
    NODE hashTable[SIZE]; // Declare the hash table
    initializeHashTable(hashTable); // Initialize all slots as empty

    // Test inserting keys into the hash table
    insert(7, hashTable);  // Should insert at hash index 0
    insert(20, hashTable); // Should insert at hash index 6
    insert(41, hashTable); // Should insert at hash index 3
    insert(31, hashTable); // Should insert at hash index 5
    insert(18, hashTable); // Should insert at hash index 4
    insert(8, hashTable);  // Should insert at hash index 1
    insert(9, hashTable);  // Should insert at hash index 2
    // Attempting additional insertions may fail due to a full table

    display(hashTable); // Display the hash table

    return 0;
}
```

rehashing

```
#include <stdio.h>
#include <stdlib.h>
#define INITIAL_SIZE 10 // Initial size of the hash table

// Structure for a node in the hash table
typedef struct node {
    int key;  // Stores the data/key
    int status; // Status of the slot: 0 = empty, 1 = occupied
} Node;

// Structure for the hash table
typedef struct {
    int size;       // Current size of the hash table
    Node *table;    // Pointer to an array of hash table nodes
} HashTable;

// Variable to track the total number of elements in the hash table
int elementCount = 0;

// Function to create a hash table with a given size
HashTable *createHashTable(int size) {
    HashTable *hash = (HashTable *)malloc(sizeof(HashTable));
    hash->size = size;
    hash->table = (Node *)malloc(size * sizeof(Node));

    // Initialize all slots as empty
    for (int i = 0; i < size; i++) {
        hash->table[i].status = 0; // Mark slot as empty
    }
    return hash;
}

// Function to destroy and free memory of the hash table
void destroyHashTable(HashTable *hash) {
    free(hash->table); // Free the hash table array
    free(hash);        // Free the hash table structure
}

// Function to rehash when the load factor exceeds a threshold
void rehash(int newKey, HashTable **hashTable) {
    int newSize = (*hashTable)->size * 2; // Double the size of the hash
table
    HashTable *newHashTable = createHashTable(newSize);

    // Reinsert all elements from the old hash table into the new hash table
    for (int i = 0; i < (*hashTable)->size; i++) {
        if ((*hashTable)->table[i].status == 1) { // If the slot is occupied
            int hashIndex, probeCount = 0;
            hashIndex = ((*hashTable)->table[i].key % newSize + probeCount)
% newSize;

            // Resolve collisions using linear probing
            while (newHashTable->table[hashIndex].status != 0) {
                probeCount++;
                hashIndex = ((*hashTable)->table[i].key % newSize +
probeCount) % newSize;
            }

            // Insert the key into the new hash table
            newHashTable->table[hashIndex].key = (*hashTable)->table[i].key;
            newHashTable->table[hashIndex].status = 1;
        }
    }

    // Free memory of the old hash table and replace it with the new one
    destroyHashTable(*hashTable);
    *hashTable = newHashTable;
}

// Function to insert a key into the hash table
void insertKey(int key, HashTable *hashTable) {
    elementCount++; // Increment the count of elements in the hash table

    // Check if rehashing is needed (load factor > 0.75)
    if (elementCount > (float)(0.75 * hashTable->size)) {
        rehash(key, &hashTable);
    }

    int hashIndex, probeCount = 0;
    hashIndex = (key % hashTable->size + probeCount) % hashTable->size;
    // Resolve collisions using linear probing
    while (hashTable->table[hashIndex].status != 0) {
        probeCount++;
        hashIndex = (key % hashTable->size + probeCount) % hashTable->size;
    }

    // Insert the key into the hash table
    hashTable->table[hashIndex].key = key;
    hashTable->table[hashIndex].status = 1; // Mark the slot as occupied
    printf("The key %d is inserted\n", key);
}

// Function to search for a key in the hash table
void searchKey(int key, HashTable *hashTable) {
    int hashIndex, probeCount = 0;
    hashIndex = (key % hashTable->size + probeCount) % hashTable->size;

    // Search the hash table using linear probing
    while (probeCount < hashTable->size) {
        if (hashTable->table[hashIndex].key == key && hashTable-
>table[hashIndex].status == 1) {
            printf("\nThe key %d is found at index %d\n", key, hashIndex);
            return;
        }
        probeCount++;
        hashIndex = (key % hashTable->size + probeCount) % hashTable->size;
    }

    // If the key is not found
    printf("\nKey %d not found\n", key);
}

// Function to display the contents of the hash table
void displayHashTable(HashTable *hashTable) {
    printf("\nHash Table:\n");
    for (int i = 0; i < hashTable->size; i++) {
        if (hashTable->table[i].status == 1) { // If the slot is occupied
            printf("Index %d: %d\n", i, hashTable->table[i].key);
        } else { // If the slot is empty
```

```
            printf("Index %d: EMPTY\n", i);
        }
    }
    printf("\n");
}

int main() {
    // Create an initial hash table with size INITIAL_SIZE
    HashTable *hashTable = createHashTable(INITIAL_SIZE);

    // Test insertions with automatic rehashing
    insertKey(5, hashTable);
    insertKey(15, hashTable);
    insertKey(25, hashTable);
    insertKey(35, hashTable);
    insertKey(45, hashTable); // Triggers rehashing
    displayHashTable(hashTable);

    // Test searching for keys
    searchKey(15, hashTable); // Should find the key
    searchKey(35, hashTable); // Should find the key
    searchKey(55, hashTable); // Should not find the key

    // Free the memory allocated for the hash table
    destroyHashTable(hashTable);
    return 0;
}
```

trie trees

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

// Trie node structure
struct TrieNode {
    struct TrieNode *children[255]; // Array to store pointers to children
nodes (one for each ASCII character)
    int isEndOfWord;               // Flag indicating the end of a valid
word
```

```
};

// Stack structure used during deletion
struct StackEntry {
    struct TrieNode *node;  // Pointer to a Trie node
    int childIndex;         // Index in the parent's children array for this
node
};
typedef struct StackEntry StackEntry;

// Global variables
char currentWord[30];       // Temporary array to store characters during
display
int currentWordLength = 0;  // Length of the current word being processed
int stackTop = -1;          // Stack's top index
StackEntry* deletionStack[30]; // Stack used for tracking nodes during
deletion

typedef struct TrieNode TrieNode;

// Function to push a Trie node and its index onto the stack
void push(TrieNode *node, int childIndex) {
    StackEntry *newEntry = (StackEntry*)malloc(sizeof(StackEntry)); //
Allocate memory for a new stack entry
    newEntry->node = node;
    newEntry->childIndex = childIndex;
    deletionStack[++stackTop] = newEntry;  // Increment stackTop and add the
new stack entry
}

// Function to pop a stack entry
StackEntry* pop() {
    return deletionStack[stackTop--];  // Return the top stack entry and
decrement `stackTop`
}

// Function to count the number of non-NULL children of a Trie node
int getChildCount(TrieNode *node) {
    int i, count = 0;
    for (i = 0; i < 256; i++) { // Iterate through all 256 possible children
```

```
        if (node->children[i] != NULL) // Increment count if a child exists
            count++;
    }
    return count;
}

// Function to create a new Trie node
TrieNode* createTrieNode() {
    int i;
    TrieNode *newNode = (TrieNode*)malloc(sizeof(TrieNode)); // Allocate
memory for a new Trie node
    for (i = 0; i < 255; i++)              // Initialize all child pointers
to NULL
        newNode->children[i] = NULL;
    newNode->isEndOfWord = 0;              // Mark end-of-word as 0 (not a
word ending yet)
    return newNode;
}

// Function to insert a word into the Trie
void insertWord(char *word, TrieNode *root) {
    int i, charIndex;
    TrieNode *currentNode = root;  // Start from the root node
    for (i = 0; word[i] != '\0'; i++) {     // Traverse through each
character of the word
        charIndex = word[i];                // Use the ASCII value of the
character as the index
        if (currentNode->children[charIndex] == NULL) // If no node exists
for this character, create one
            currentNode->children[charIndex] = createTrieNode();
        currentNode = currentNode->children[charIndex]; // Move to the child
node
    }
    currentNode->isEndOfWord = 1;  // Mark the end of the word
}

// Recursive function to display all words in the Trie
void displayWords(TrieNode *root) {
    TrieNode *currentNode = root;
    int i, j;
    for (i = 0; i < 255; i++) {             // Traverse all possible
children
        if (currentNode->children[i] != NULL) { // If a child exists
            currentWord[currentWordLength++] = i; // Add the character to
the currentWord array
            if (currentNode->children[i]->isEndOfWord == 1) { // If it's the
end of a valid word, print it
                printf("\n");
                for (j = 0; j < currentWordLength; j++)
                    printf("%c", currentWord[j]);
            }
            displayWords(currentNode->children[i]); // Recursively display
all words under this child
        }
    }
    currentWordLength--;                  // Backtrack after exploring a
branch
    return;
}

// Function to search for a word in the Trie
void searchWord(char *word, TrieNode *root) {
    int i, charIndex;
    TrieNode *currentNode = root;          // Start from the root node
    for (i = 0; word[i] != '\0'; i++) {    // Traverse through each
character of the word
        charIndex = word[i];
        if (currentNode->children[charIndex] == NULL) { // If no child
exists for the character, the word doesn't exist
            printf("\nWord not found");
            return;
        }
        currentNode = currentNode->children[charIndex]; // Move to the child
node
    }
    if (currentNode->isEndOfWord == 1)                  // Check if this is
the end of a valid word
        printf("\nWord found");
    else
        printf("\nWord not found");
    return;
}

// Function to delete a word from the Trie
void deleteWord(char *word, TrieNode *root) {
    int charIndex, i;
    StackEntry *stackEntry;
    TrieNode *currentNode = root;

    // Traverse the word and push nodes onto the stack
    for (i = 0; word[i] != '\0'; i++) {
        charIndex = word[i];
        if (currentNode->children[charIndex] == NULL) { // If any character
is missing, the word doesn't exist
            printf("\nWord not found");
            return;
        }
        push(currentNode, charIndex);                  // Push the current
node and index to the stack
        currentNode = currentNode->children[charIndex];
    }
    currentNode->isEndOfWord = 0;                      // Unmark the end of
the word

    // If the node has children, we cannot delete it
    if (getChildCount(currentNode) >= 1)
        return;

    // Otherwise, backtrack and delete unnecessary nodes
    stackEntry = pop();
    currentNode = stackEntry->node;
    charIndex = stackEntry->childIndex;
    while (getChildCount(currentNode) <= 1 && currentNode->isEndOfWord == 0)
{ // Keep deleting as long as the node is not needed
        free(currentNode->children[charIndex]);       // Free the child node
        currentNode->children[charIndex] = NULL;      // Remove the
reference to the child
        stackEntry = pop();
        currentNode = stackEntry->node;
        charIndex = stackEntry->childIndex;
    }
}

// Main function: User interface for Trie operations
void main() {
    TrieNode *root;
    int choice;
    char word[30];
    root = createTrieNode();              // Initialize the root node

    do {
        printf("\nEnter your choice: 1.Insert 2.Display 3.Search 4.Delete
5.Exit: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\nEnter the word to be inserted: ");
                scanf("%s", word);
                insertWord(word, root);   // Insert the word
                break;
            case 2:
                currentWordLength = 0;    // Reset the global
`currentWordLength` variable
                displayWords(root);       // Display all words
                break;
            case 3:
                printf("\nEnter the word to be searched: ");
                scanf("%s", word);
                searchWord(word, root);   // Search for the word
                break;
            case 4:
                printf("\nEnter the word to be deleted: ");
                scanf("%s", word);
                deleteWord(word, root);   // Delete the word
                break;
            default:
                return;                   // Exit the program
        }
    } while (choice);                     // Repeat until the user exits
```

```
 }
```