Singly Linked List

# Definition of a Linked List:

A linked list is a linear data structure in which elements, called nodes, are stored in a sequential manner. Each node contains two components:

1. Data: The actual value or information stored in the node.

2. Pointer: A reference to the next node in the sequence.

# Types of Linked Lists:

- Singly Linked List: Each node points to the next node in the sequence. The last node points to NULL, indicating the end of the list.

- Doubly Linked List: Each node contains two pointers, one pointing to the next node and one pointing to the previous node.

- Circular Linked List: The last node points back to the first node, forming a circular structure.

# Node structure:

struct Node {

int data;

struct Node\* next;

};

# Visual Representation:

Head -> [Data | Next] -> [Data | Next] -> [Data | Next] -> NULL

# Example Use Cases:

- Implementing stacks and queues

- Dynamic memory management

- Representing complex data structures like graphs and trees

# Insertion Operations in C:

### 1. Insertion at the Beginning:

void insertAtBeginning(struct Node\*\* head, int newData) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = newData;

newNode->next = \*head;

\*head = newNode;

}

### 2. Insertion at the End:

void insertAtEnd(struct Node\*\* head, int newData) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* last = \*head;

newNode->data = newData;

newNode->next = NULL;

if (\*head == NULL) {

\*head = newNode;

return;

}

while (last->next != NULL) {

last = last->next;

}

last->next = newNode;

}

### 3. Insertion at a Given Position:

void insertAtPosition(struct Node\*\* head, int newData, int position) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = newData;

if (position == 0) {

newNode->next = \*head;

\*head = newNode;

return;

}

struct Node\* current = \*head;

for (int i = 0; current != NULL && i < position - 1; i++) {

current = current->next;

}

if (current == NULL) {

printf("Position is greater than the number of nodes.\n");

return;

}

newNode->next = current->next;

current->next = newNode;

}

# Deletion Operations in C:

### 1. Deletion from the Beginning:

void deleteFromBeginning(struct Node\*\* head) {

if (\*head == NULL) {

printf("List is empty.\n");

return;

}

struct Node\* temp = \*head;

\*head = (\*head)->next;

free(temp);

}

### 2. Deletion from the End:

void deleteFromEnd(struct Node\*\* head) {

if (\*head == NULL) {

printf("List is empty.\n");

return;

}

struct Node\* temp = \*head;

if ((\*head)->next == NULL) {

free(\*head);

\*head = NULL;

return;

}

while (temp->next->next != NULL) {

temp = temp->next;

}

free(temp->next);

temp->next = NULL;

}

### 3. Deletion from a Given Position:

void deleteFromPosition(struct Node\*\* head, int position) {

if (\*head == NULL) {

printf("List is empty.\n");

return;

}

struct Node\* temp = \*head;

if (position == 0) {

\*head = temp->next;

free(temp);

return;

}

for (int i = 0; temp != NULL && i < position - 1; i++) {

temp = temp->next;

}

if (temp == NULL || temp->next == NULL) {

printf("Position does not exist.\n");

return;

}

struct Node\* next = temp->next->next;

free(temp->next);

temp->next = next;

}

# Traversal:

void printList(struct Node\* node) {

while (node != NULL) {

printf("%d -> ", node->data);

node = node->next;

}

printf("NULL\n");

}

# Searching:

bool search(struct Node\* head, int key) {

struct Node\* current = head;

while (current != NULL) {

if (current->data == key) {

return true;

}

current = current->next;

}

return false;

}

# Example Main Function:

Here's an example main function demonstrating the above operations in C:

int main() {

struct Node\* head = NULL;

// Insertion operations

insertAtEnd(&head, 10);

insertAtBeginning(&head, 5);

insertAtEnd(&head, 20);

insertAtPosition(&head, 15, 2);

// Print list

printf("Linked List: ");

printList(head);

// Search operation

int key = 15;

if (search(head, key)) {

printf("%d is found in the list.\n", key);

} else {

printf("%d is not found in the list.\n", key);

}

// Deletion operations

deleteFromBeginning(&head);

deleteFromEnd(&head);

deleteFromPosition(&head, 1);

// Print list again

printf("Linked List after deletions: ");

printList(head);

return 0;

}

# Memory Management:

Memory management is crucial when dealing with singly linked lists in C, as it involves dynamic memory allocation and deallocation to ensure efficient use of resources and avoid memory leaks. Here’s a detailed guide on how to manage memory in a singly linked list:

### Dynamic Memory Allocation:

When creating new nodes, you need to allocate memory dynamically using functions from the stdlib.h library, primarily malloc and free.

### 1. Allocating Memory for a Node:

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL) {

printf("Memory allocation failed\n");

return;

}

newNode->data = someData;

newNode->next = NULL;

- malloc: Allocates a specified number of bytes and returns a pointer to the allocated memory.

- sizeof: Ensures that the correct amount of memory is allocated for a Node.

### Memory Deallocation:

To prevent memory leaks, it's important to deallocate memory once it is no longer needed. This is done using the free function.

### 1. Freeing a Single Node:

free(nodePointer);

# Space Complexity

The space complexity for a singly linked list primarily depends on the number of elements (nodes) in the list.

- Space Complexity: O(n)

- Each node in the list requires a fixed amount of space for storing data and a pointer to the next node.

- For (n) nodes, the total space used is proportional to (n), thus O(n).

# Time Complexity:

## Insertion Operations:

### 1. Insertion at the Beginning:

- Time Complexity: O(1)

- Inserting a new node at the beginning involves updating the next pointer of the new node to point to the current head and then updating the head to point to the new node.

- This operation takes a constant amount of time regardless of the list size.

### 2. Insertion at the End:

- Time Complexity: O(n)

- To insert a node at the end, you need to traverse the entire list to find the last node (which takes O(n) time), and then update the next pointer of the last node.

- If a tail pointer is maintained, this can be reduced to O(1).

### 3. Insertion at a Given Position:

- Time Complexity: O(n)

- To insert at a specific position, you must traverse the list to reach the position (which takes O(k) time where (k) is the position), and then update the pointers.

- In the worst case, (k) can be (n), making the time complexity (O(n).

## Deletion Operations:

### 1. Deletion from the Beginning:

- Time Complexity: O(1)

- Deleting the first node involves updating the head pointer to point to the next node.

- This operation takes constant time.

### 2. Deletion from the End:

- Time Complexity: O(n)

- To delete the last node, you need to traverse the list to find the second-to-last node (which takes O(n) time), and then update its next pointer to NULL.

- If a tail pointer is maintained, you also need to update the tail pointer, but the traversal still takes O(n)time.

### 3. Deletion from a Given Position:

- Time Complexity: O(n)

- To delete a node from a specific position, you must traverse the list to reach the position (which takes O(k) time where (k) is the position), and then update the pointers.

- In the worst case, (k) can be (n), making the time complexity O(n).

## Search Operation:

### Search for a Value:

- Time Complexity: O(n)

- Searching for a value involves traversing the list from the head to the end, checking each node's data.

- In the worst case, the value is not found, and the traversal takes O(n) time.

# Practical Applications include:

1. Implementation of Other Data Structures like stacks and queues

2. Dynamic Memory Allocation

3. Graph Representations

4. Navigation Systems

5. File Systems

6. Operating Systems

7. Text Editors

8. Music and Media Playlists

9. Simulation and Modeling

10. Web Browsers

11. Blockchain Technology

# Example in C: Implementing a Simple Playlist

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct Song {

char title[50];

struct Song\* next;

};

void addSong(struct Song\*\* head, const char\* title) {

struct Song\* newSong = (struct Song\*)malloc(sizeof(struct Song));

strcpy(newSong->title, title);

newSong->next = \*head;

\*head = newSong;

}

void printPlaylist(struct Song\* head) {

struct Song\* current = head;

while (current != NULL) {

printf("%s\n", current->title);

current = current->next;

}

}

void freePlaylist(struct Song\* head) {

struct Song\* current = head;

struct Song\* next;

while (current != NULL) {

next = current->next;

free(current);

current = next;

}

}

int main() {

struct Song\* playlist = NULL;

addSong(&playlist, "Song 1");

addSong(&playlist, "Song 2");

addSong(&playlist, "Song 3");

printf("Playlist:\n");

printPlaylist(playlist);

freePlaylist(playlist);

return 0;

}

# Advantages of singly linked list:

1. Dynamic Size

2. Ease of Insertion and Deletion

3. Efficient Memory Utilization

4. Simple Implementation

5. Flexibility

# Disadvantages of singly linked list:

1. No Random Access

2. Increased Memory Usage

3. Pointer Overhead

4. Cache Performance

5. Complexity in Operations