Module (2)

Linked List Overview

Linked List Overview

- **Linear Data Structure**: Elements are arranged in a sequence: first, second, third, etc.
- **Non-Contiguous Memory**: Elements are stored at arbitrary locations in memory.
- **Heterogeneous Data Structure**: Each element/node contains multiple fields.
- **Dynamic Data Structure**: Memory allocation can change at runtime (nodes can be added or removed).
- **Sequential Access**: Nodes must be accessed in order from the first node onwards.

A type of data structure where each element is connected to its previous and next element, forming a sequence.

Structure of a Linked List

- **Node**: Basic unit of a linked list.
  - **Information Field**: Stores the actual data.
  - **Address Field**: Stores the address of the next node.
- **Head**: Pointer to the first node in the list. Essential for accessing the list.

Node Implementation in C

- **Defining a Node Structure**:

  ```c
  struct Node {
      int info;
      struct Node* next;
  };
  ```

  - `info`: Stores integer data.
  - `next`: Pointer to the next node.

Example of a Linked List

- **Creating Nodes**:

```c
struct Node* head = (struct Node*)malloc(sizeof(struct Node));
head->info = 3;
head->next = (struct Node*)malloc(sizeof(struct Node));
head->next->info = 1;
head->next->next = (struct Node*)malloc(sizeof(struct Node));
head->next->next->info = 7;
head->next->next->next = NULL;
```

- `head` points to the first node.
- `head->next` points to the second node.
- `head->next->next` points to the third node.
- The last node's `next` field is set to `NULL`.

Accessing Nodes

- **Accessing Data**:
  - `head->info`: Data of the first node.
  - `head->next->info`: Data of the second node.
  - `head->next->next->info`: Data of the third node.
- **Accessing Next Nodes**:
  - `head->next`: Address of the second node.
  - `head->next->next`: Address of the third node.

Types of Linked Lists

1. **Singly Linked List**:
   - Each node stores the address of only the next node.
   - Unidirectional: Can only move forward.
2. **Doubly Linked List**:
   - Each node stores the address of both the next and the previous nodes.
   - Bidirectional: Can move both forward and backward.
   - **Node Structure**:

     ```c
     struct Node {
         int info;
         struct Node* next;
         struct Node* prev;
     };
     ```

3. **Circular Linked List**:
   - The last node points back to the first node, forming a circle.
   - Can be singly or doubly circular linked list.

- Allows for continuous traversal from last to first and vice versa.

Summary

- Linked lists are a fundamental data structure offering flexibility in memory usage.
- Different types of linked lists (singly, doubly, circular) provide various traversal capabilities.
- Understanding the node structure and how to manipulate it is crucial for implementing linked lists in C.

Linked List Implementation

Introduction

- Linked lists can be implemented using:
  - Dynamic memory allocation
  - Static memory allocation

Dynamic Memory Allocation

- Memory allocation occurs at the time of creating a node.
- **Advantages:**
  - Space-efficient: Memory is allocated only when a node is created.
  - Unlimited growth: Number of nodes can grow as long as system memory permits.
- **Implementation:**

  - Define a node using a `struct`.
  - Use `malloc` function to allocate memory for each node.
  - Which function is used for dynamic memory allocation in C?

  malloc()
  Correct
  In C programming, malloc() stands for "memory allocation." It is used to dynamically allocate a specified number of bytes of memory during the execution of a program.
  free()
  calloc()
  Correct
  calloc() also allocates memory like malloc(), but it initializes the allocated memory to zero.
  realloc()

Example:

```
struct Node {
    int data;
    struct Node* next;
};

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

Static Memory Allocation

- Memory is allocated before creating the linked list.
- **Disadvantages:**
  - Space inefficient: Pre-allocated space may not be fully used.
  - Limited size: Number of nodes is limited by the size of the pre-defined array.
- **Implementation:**
  - Define nodes within an array.
  - Use integers to represent the index of the next node instead of pointers.

Example:

```
struct Node {
    int data;
    int next;
};

struct Node array[10];  // Array size limits the linked list to 10 elements.
```

Comparison: Dynamic vs. Static Memory Allocation

- **Dynamic:**
  - Uses pointers for next node.
  - More natural and efficient.
- **Static:**
  - Uses array indices for next node.
  - Limited by predefined array size.
  - Less efficient and flexible.

Detailed Static Memory Allocation Example:

- **Node Structure:**

    - Two fields: information and next.
- **Array Representation:**
    - Two-dimensional array: One column for information, one for the next index.

Example:

```
struct Node {
    int data;
    int next;
};

struct Node array[10];
```

- **Storing Nodes:**
    - First node at index 5.
    - Second node at index 2, stored in the next field of the first node.
    - Third node at index 8, stored in the next field of the second node.

Traversing the List:

- Start from head.
- Use stored indices to move from one node to another.
- Use an invalid number (e.g., -1) to mark the end of the list.

Code Implementation:

```
#include <stdio.h>

#define SIZE 10

struct Node {
    int data;
    int next;
};

struct Node array[SIZE];

int main() {
    // Example: Initializing nodes
    array[5].data = 3;
    array[5].next = 2;

    array[2].data = 5;
    array[2].next = 8;
```

```c
    array[8].data = 7;
    array[8].next = -1;  // End of the list

    int head = 5;
    int current = head;
    while(current != -1) {
        printf("%d -> ", array[current].data);
        current = array[current].next;
    }
    printf("NULL\n");

    return 0;
}
```

Conclusion

- Linked lists are generally implemented using dynamic memory allocation due to their efficiency and flexibility.
- Static memory allocation is less efficient and limits the number of nodes by the predefined array size.

Notes on Implementing Singly Linked Lists

Introduction to Singly Linked Lists

- **Definition**: A sequence of nodes where each node stores the address of the next node.
- **Structure**:
  - Node contains data and a pointer to the next node.
  - Dynamic memory allocation is used for efficiency and flexibility.
- **Properties**:
  - Allows only forward traversal.
  - Must always store the address of the first node (head).
  - Sequential access only (no random access).

Operations on Singly Linked Lists

1. **Creating a Linked List**
   - **Create Operation**: Initializes an empty linked list.
     - Declare a pointer variable `head` to store the address of the first node.
     - Example: Declaring `head` initializes an empty linked list.
2. **Traversal of Linked List**
   - **Traversal Operation**: Visits all nodes from the first to the last.

        - Define a pointer variable `ptr`.
        - Initialize `ptr` with the address of the first node (`ptr = head`).
        - Iterate through the list:
            - Print the value of the current node.
            - Move `ptr` to the next node (`ptr = ptr->next`).
            - Continue until `ptr` is `NULL`.
    - **Time Complexity**: O(n)
    - **Space Complexity**: O(n)
3. **Search in Linked List**
    - **Search Operation**: Finds an element in the linked list.
        - Use a pointer to traverse the list.
        - Check each node's data for the target value.
        - Return if the element is found; otherwise, continue to the next node.
        - Terminate if the end of the list is reached without finding the element.
    - **Time Complexity**:
        - Best case: O(1) (if element is in the first node)
        - Worst case: O(n) (if element is in the last node or not found)
        - Average case: O(n)
    - **Space Complexity**: O(n)

Detailed Steps and Functions

1. **Creating an Empty Linked List**

```
   struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL; // Creates an empty linked list
```

2. **Traversal Function**

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

void traverse(struct Node *head) {*
struct Node ptr = head;
while (ptr != NULL) {
printf("%d ", ptr->data);

ptr = ptr->next;
}
}

````
3. **Search Function**
```C
#include <stdio.h>

// Custom bool type definition
typedef enum { false, true } bool;

bool search(struct Node* head, int value) {
 struct Node* ptr = head;
 while (ptr != NULL) {
     if (ptr->data == value) {
         printf("Found\n");
         return true;
     }
     ptr = ptr->next;
 }
 printf("Not Found\n");
 return false;
}
````

Summary

- A singly linked list is a sequential data structure allowing only forward traversal.
- Essential operations include creation, traversal, and search.
- **Traversal** and **search** operations involve iterating through all nodes.
- **Time Complexity** for both traversal and search operations is O(n).
- **Space Complexity** is O(n), determined by the number of nodes in the list.

Insertion Operation in Singly Linked List

Introduction

- Discussing insertion operation in a singly linked list.
- Various scenarios:
  - Insertion as the first node.
````

- Insertion as the last node.
- Insertion at any position.

Insertion as the First Node

1. **Create a New Node**
    - Define a pointer variable for the new node.
    - Allocate memory using `malloc`.
    - Store the new data into the new node.
2. **Establish Connection**
    - Set the `next` field of the new node to point to the current first node.
    - Update the head pointer to point to the new node.
3. **Implementation in a Function**
    - Function `insertFirst` takes the address of the header and the element to insert.
    - Create a new node and assign the required value.
    - Connect the new node to the existing linked list.
    - Update the head pointer.
4. **Complexity**
    - Time Complexity: O(1)
    - Space Complexity: O(n)

Insertion as the Last Node

1. **Create a New Node**
    - Allocate memory for a new node.
    - Assign the new value and set `next` to null.
2. **Traverse to the Last Node**
    - Scan the linked list until reaching the last node.
    - Update a pointer variable to move through the list.
3. **Connect Nodes**
    - Set the `next` field of the last node to point to the new node.
4. **Implementation in a Function**
    - Function `insertLast` takes the address of the first node and the element to insert.
    - Create a new node and assign the required value.
    - Traverse to the last node.
    - Connect the new node as the last node.
5. **Complexity**
    - Time Complexity: O(n)
    - Space Complexity: O(n)

Insertion at Any Position

1. **Create a New Node**
    - Allocate memory for a new node.
    - Assign the new value and set `next` to null.
2. **Traverse to the Insertion Position**
    - Scan the linked list up to the position minus one.
3. **Connect Nodes**
    - Establish connection from the previous node to the new node.
    - Then connect the previous node to the new node.
4. **Implementation in a Function**
    - Function `insertPosition` takes the address of the first node, the position, and the element to insert.
    - Create a new node and assign the required value.
    - Traverse to the insertion position.
    - Connect the new node at the desired position.
5. **Complexity**
    - Time Complexity: O(n)
    - Space Complexity: O(n)

Conclusion

- Time and space complexities vary based on the insertion position.
- Understanding the sequence of connections is crucial for correct insertion.
- Each insertion method provides a different set of complexities based on traversal requirements.

Linked List Deletion Operations

1. **Introduction:**
    - Discussing delete operations in linked lists.
    - Three scenarios: delete the first node, delete the last node, delete a node at any position.
2. **Delete the First Node:**
    - Update the head pointer to the next node.
    - Function to achieve this:
        - Takes the address of the first node (head pointer).
        - Updates the head pointer to head pointer next.
    - Time Complexity: Θ(1)
    - Space Complexity: Θ(n)
3. **Delete the Last Node:**

- Scan the entire linked list till the last node.
- Maintain two pointer variables: current and previous.
- Traverse from the first node to the last.
- When current points to the last node, previous points to the last but one node.
- Disconnect the last node by assigning null to previous next.
- Function to achieve this:
  - Takes the address of the first node.
  - Uses two pointers: current and previous.
  - Traverses until current next is null, then sets previous next to null.
- Time Complexity: O(n)
- Space Complexity: O(n)

4. **Delete a Node at a Specific Position:**
   - Maintain current and previous pointers.
   - Traverse until reaching the desired position.
   - Update the previous next to the current's next.
   - Function to achieve this:
     - Traverse to the node to delete and update previous next.
   - Time Complexity: O(n)
   - Space Complexity: O(n)
5. **Summary:**
   - Deleting the first node: Update head pointer.
   - Deleting the last node: Traverse to the end and disconnect.
   - Deleting a node at a specific position: Traverse to the position and update pointers.
   - Time and space complexities are both O(n) for all deletion operations due to the need to traverse the entire list and maintain node storage.
6. **Conclusion:**
   - Efficient deletion operations are crucial in linked list management.
   - Understanding the algorithms and their complexities is essential for effective implementation.

Doubly Linked Lists (DLL)

1. **Introduction to Doubly Linked Lists (DLL)**
   - **Review of Singly Linked Lists (SLL)**
     - Unidirectional traversal (head to tail)
   - **DLL Overview**
     - Bidirectional traversal (head to tail and tail to head)

    - Each node contains two pointers: one to the next node and one to the previous node.

2. **Structure of a DLL Node**
   - Each node has:
     - **Information field**: to store data.
     - **Next pointer**: to store the address of the next node.
     - **Previous pointer**: to store the address of the previous node.
   - Example structure:

   ```
   struct Node {
       int data;
       Node* next;
       Node* prev;
   };
   ```

3. **Properties of DLL**
   - Linear data structure: Nodes are arranged sequentially.
   - Sequential access: To visit a node, you must traverse from a starting node (head or tail).
   - Traversal:
     - Forward traversal: using `next` pointers.
     - Backward traversal: using `prev` pointers.
   - Memory allocation:
     - Can be implemented using static or dynamic memory allocation.
     - Dynamic memory allocation is more efficient and natural for linked lists.

4. **Creating a DLL Node and Building the List**
   - **Node Creation Example:**

   ```
   Node* newNode = new Node();
   newNode->data = 3;
   newNode->next = nullptr;
   newNode->prev = nullptr;
   ```

   - **Maintaining Pointers:**
     - `head` points to the first node.
     - `tail` points to the last node.
     - Initially, both `head` and `tail` point to the single node created.

5. **Step-by-Step DLL Node Insertion Example**
   - **First Node Creation:**
     - Allocate memory for the node.

        - Set data field.
        - Set `next` and `prev` pointers to `nullptr`.
        - Assign `head` and `tail` to point to this node.

        ```
        Node* head = new Node();
        head->data = 3;
        head->next = nullptr;
        head->prev = nullptr;
        Node* tail = head;
        ```

    - **Second Node Creation:**
        - Allocate memory for the new node.
        - Set data field.
        - Set `next` pointer to `nullptr`.
        - Set `prev` pointer to the current `tail`.
        - Update `tail->next` to point to the new node.
        - Move `tail` to the new node.

        ```
        Node* newNode = new Node();
        newNode->data = 1;
        newNode->next = nullptr;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
        ```

    - **Third Node Creation:**
        - Allocate memory for the new node.
        - Set data field.
        - Set `next` pointer to `nullptr`.
        - Set `prev` pointer to the current `tail`.
        - Update `tail->next` to point to the new node.
        - Move `tail` to the new node.

        ```
        Node* newNode = new Node();
        newNode->data = 7;
        newNode->next = nullptr;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
        ```

6. **Benefits of Maintaining Both `head` and `tail` Pointers**
    - **Efficient Insertions at End:**

    - Directly using `tail` reduces insertion time complexity from O(n) to O(1).
  - **Bidirectional Traversal:**
    - Forward traversal from `head`.
    - Backward traversal from `tail`.
7. **Pointer Manipulations in DLL**
   - Flexibility in pointer usage:
     - Can use either `head` or `tail` to manage node connections.
     - Example of using `head` for forward connection and `tail` for backward connection:

```
Node* newNode = new Node();
newNode->data = value;
newNode->next = nullptr;
newNode->prev = tail;
tail->next = newNode;
tail = newNode;
```

These notes cover the main concepts and detailed steps for understanding and implementing doubly linked lists, following the structure and order presented in the transcript.

Notes on Doubly Linked List Operations:

1. **Introduction to Doubly Linked List Operations:**
   - Operations similar to singly linked list.
   - Create function: Create an empty doubly linked list.
   - Insert function: Insert at first, last, or any position.
   - Delete function: Delete first, last, or any position.
   - Search function: Search for an element.
   - Traversal function: Traverse from first to last or last to first.
2. **Implementation of Basic Operations:**
   - Creation of an empty doubly linked list.
   - Implementation left as an assignment.
3. **Insert Operation:**
   - Three scenarios: Insert at first, last, or any position.
   - Detailed implementation of inserting a node as the first node.
4. **Insertion at First Node:**
   - Create a new node.
   - Assign value and null pointers.
   - Connect new node with the previous first node.

   - Update head pointer to new node.
   - Time complexity: O(1).
   - Space complexity: O(n).
5. **Insertion at Last Node:**
   - Create a new node.
   - Assign value and null pointers.
   - Connect new node with the previous last node.
   - Update tail pointer to new node.
   - Time complexity: O(1).
   - Space complexity: O(n).
6. **Optimizing Time Complexity:**
   - Use of tail pointer reduces traversal.
   - Direct access to the last node.
7. **Conclusion:**
   - Insertion operations simplified with doubly linked lists.
   - Constant time complexity achieved for insertions.
   - Space complexity remains linear.

Notes on Doubly Linked List Insertion at Any Position:

1. **Introduction to Insertion at Any Position:**
   - Need to insert a node at any random position.
   - Three possible cases discussed:
     - Case 1: Position <= half of total nodes, scan from head.
     - Case 2: Position > half of total nodes, scan backward from tail.
     - Case 3: Simultaneous scanning forward and backward.
2. **Approach for Case 3: Simultaneous Scanning:**
   - Two pointers used: `head current` (forward) and `tail current` (backward).
   - Creation of new node using malloc.
   - Assigning head and tail pointers to head and tail respectively.
   - Simultaneous scanning using while loop:
     - Incrementing counter while scanning.
     - Moving `head current` forward (`head_current = head_current->next`).
     - Moving `tail current` backward (`tail_current = tail_current->prev`).
3. **Establishing Required Connection:**
   - After finding the position, connecting new node with the node at that position.
   - Checking if position is in first or second half of the list.
   - Establishing connection based on position:
     - If in first half, using `head current`.

    - If in second half, using `tail current`.
4. **Completing Insertion Process:**
   - Connecting new node with node at the position.
   - Adjusting connections based on position.
   - Illustration of connection establishment for both first and second halves of the list.
5. **Defining Function for Insertion:**
   - Defining function `insertPosition` with parameters:
     - Address of first node,
     - Address of tail node,
     - Element to insert,
     - Position to insert,
     - Total number of nodes in existing list.
6. **Time and Space Complexity Analysis:**
   - Time Complexity: O(n) since scanning up to middle nodes.
   - Space Complexity: O(n) due to storage of nodes.
7. **Conclusion:**
   - Algorithm scans simultaneously from both sides.
   - Time complexity is O(n) and space complexity is O(n).

Deleting a Node in a Doubly Linked List

Introduction

- Deleting a node from a doubly linked list involves three possible scenarios:
  1. Deleting the first node.
  2. Deleting the last node.
  3. Deleting a node at any position.

Deleting the First Node

1. **Initial State**:
   - Head points to the first node.
   - This node will be deleted.
2. **Steps**:
   - Update the head pointer to point to the second node:

     ```
     head = head->next;
     ```

   - Set the `previous` field of the new head (second node) to `NULL`:

```
head->previous = NULL;
```

3. **Result**:
   - The first node becomes a dangling node and is effectively deleted.
4. **Complexity**:
   - Time Complexity: (O(1))
   - Space Complexity: (O(n))

Deleting the Last Node

1. **Initial State**:
   - Tail points to the last node.
   - This node will be deleted.
2. **Steps**:
   - Update the `next` field of the second last node to `NULL`:

   ```
   tail->previous->next = NULL;
   ```

   - Update the tail pointer to point to the second last node:

   ```
   tail = tail->previous;
   ```

3. **Result**:
   - The last node is deleted.
4. **Complexity**:
   - Time Complexity: (O(1))
   - Space Complexity: (O(n))

Deleting a Node at Any Position

1. **Initial State**:
   - We need to delete a node at a given position, say position 3.
2. **Approaches**:
   - **Case 1**: Scan from the first node forward.
   - **Case 2**: Scan from the last node backward.
   - **Case 3**: Scan simultaneously from both ends.
3. **Preferred Method (Case 3)**:
   - **Step 1**: Initialize two pointers:

```
Node* headCurrent = head;
Node* tailCurrent = tail;
```

- **Step 2:** Simultaneously scan forward and backward to locate the node:

```
int count = 1;
while (count < position) {
    headCurrent = headCurrent->next;
    tailCurrent = tailCurrent->previous;
    count++;
}
```

- **Step 3:** Delete the node:
  - If node is in the first half:

    ```
    headCurrent->next = headCurrent->next->next;
    headCurrent->next->next->previous = headCurrent;
    ```

  - If node is in the second half:

    ```
    tailCurrent->previous = tailCurrent->previous->previous;
    tailCurrent->previous->previous->next = tailCurrent;
    ```

4. **Complexity:**
   - Time Complexity: (O(n/2) = O(n))
   - Space Complexity: (O(n))

Summary

- **First Node Deletion**: Update head and head's previous field.
- **Last Node Deletion**: Update tail and tail's next field.
- **Arbitrary Position Deletion**: Use dual scanning to find and delete the node efficiently.

Session Overview: Circular Linked List

1. Introduction to Linked Lists:

- Recap of singly linked list and doubly linked list.
- Discussion on the operations: create, insert, delete, search, traverse.

2. Circular Linked List:

- Definition and explanation of circular linked list.
- Advantages of circular linked list:
    - No distinct head node, any node can be a starting point.
    - Circular traversal possible.

3. Circular Singly Linked List:

- **Structure Definition:**
    - Node structure: integer data and a pointer to the next node.
- **Creating a Circular Singly Linked List with One Node:**
    - Define node structure.
    - Allocate memory for the node.
    - Set the node's `next` pointer to point to itself, creating the circular connection.
- **Insertion in Circular Singly Linked List:**
    - Define node structure and create the first node.
    - Create and connect the new node by adjusting pointers.
    - Ensure the circular connection is maintained.
- **Deletion in Circular Singly Linked List:**
    - Identify and delete the target node.
    - Adjust the pointers of the surrounding nodes to maintain the circular connection.
- **Traversal in Circular Singly Linked List:**
    - Traverse the list until reaching the starting node again.
    - Condition: `while (pointer->next != head)`

4. Circular Doubly Linked List:

- **Structure Definition:**
    - Node structure: integer data, a pointer to the next node, and a pointer to the previous node.
- **Creating a Circular Doubly Linked List with One Node:**
    - Define node structure.
    - Allocate memory for the node.
    - Set the node's `next` and `previous` pointers to point to itself, creating the circular connection.
- **Insertion in Circular Doubly Linked List:**
    - Define node structure and create the first node.
    - Allocate memory for the new node.
    - Adjust pointers to connect the new node ensuring both forward and backward connections.

- **Deletion in Circular Doubly Linked List:**
  - Identify and delete the target node.
  - Adjust the pointers of the surrounding nodes to maintain the circular connections.
  - Update the head pointer if necessary.
- **Traversal in Circular Doubly Linked List:**
  - Similar to singly circular linked list but with the option to traverse forward or backward.
  - Condition: `while (pointer->next != head)` or `while (pointer->previous != head)`

Circular Singly Linked List Implementation:

- **Node Structure:**

```
struct Node {
    int data;
    struct Node* next;
};
```

- **Create a Node:**

```
Node* head = (Node*)malloc(sizeof(Node));
head->data = 7;
head->next = head; // Circular connection
```

- **Insert a Node:**

```
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->data = 3;
newNode->next = head->next; // Connect to head's next
head->next = newNode; // Connect head to new node
```

- **Delete a Node:**

```
Node* temp = head;
while(temp->next != head) {
    temp = temp->next;
}
Node* delNode = head;
temp->next = head->next; // Remove head node
head = head->next; // Update head
free(delNode);
```

- **Traverse a Circular Singly Linked List:**

```
Node* temp = head;
do {
    printf("%d ", temp->data);
    temp = temp->next;
} while(temp != head);
```

Circular Doubly Linked List Implementation:

- **Node Structure:**

```
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
```

- **Create a Node:**

```
Node* head = (Node*)malloc(sizeof(Node));
head->data = 7;
head->next = head; // Circular connection
head->prev = head; // Circular connection
```

- **Insert a Node:**

```
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->data = 3;
newNode->next = head->next;
newNode->prev = head;
head->next->prev = newNode;
head->next = newNode;
```

- **Delete a Node:**

```
Node* delNode = head;
head->prev->next = head->next;
head->next->prev = head->prev;
head = head->next;
free(delNode);
```

- **Traverse a Circular Doubly Linked List:**

```
Node* temp = head;
do {
    printf("%d ", temp->data);
    temp = temp->next;
} while(temp != head);
```

These notes cover the main points from the transcript in a structured manner, including definitions, creation, insertion, deletion, and traversal of circular linked lists in both singly and doubly linked contexts.

Notes on Linked Lists with Dedicated Head Node

1. Introduction to Linked Lists

- **Types of Linked Lists**: Singly Linked List, Doubly Linked List, Circular Linked List.
- **Dedicated Head Node**: A node specifically for the head, often referred to as a dummy node or header node.

2. Issues with Previous Implementations

- **Pointer to Pointer Variables**:
  - Necessary for operations modifying the head pointer (e.g., inserting or deleting the first node).
  - Ensures changes to the head pointer in functions are reflected in the caller function.

3. Inserting Nodes in Singly Linked List

- **Without Dedicated Head Node**:
  - Directly manipulate the head pointer.
  - Use pointer to pointer to reflect changes in the caller function.
- **With Dedicated Head Node**:
  - Introduce a dummy node (header node) whose address is stored in the head pointer.
  - Header node simplifies insertion and deletion operations without modifying the head pointer directly.

  Insertion Steps:
  - Create a new node.
  - Establish connections between new node and the rest of the list.
  - Update the next field of the header node to point to the new node.

4. Example Implementation of Insertion

- **Create a Node**:
  - Allocate space using `malloc`.
  - Assign value to the new node.
  - Link new node with the next node (if any).
  - Update the header node to point to the new node.

5. Creating an Empty Linked List with a Dedicated Head Node

- **Empty List Initialization**:

  - Create a dummy node.
  - Head pointer points to the dummy node.
  - Dummy node's next field is NULL.

  Code Example:

```
// Create header node
Node* header = (Node*)malloc(sizeof(Node));
header->next = NULL; // Initialize next as NULL
```

6. Advantages of Dedicated Head Node

- **Uniform Treatment**:
  - Simplifies insertions and deletions.
  - No need to modify the head pointer directly.
  - Useful information can be stored in the dummy node (e.g., count of nodes).

7. Application to Different Linked Lists

- **Singly Linked List**:
  - Header node points to the first real node.
- **Circular Linked List**:
  - Header node is part of the circular structure.
  - Last node points to the header node.
- **Doubly Linked List**:
  - Header node helps in managing forward and backward links.
  - Can also be circular by linking the last node to the header node.

8. Summary

- Using a dedicated head node or dummy node simplifies the implementation of linked lists.
- It provides a uniform approach to handle insertions and deletions.
- Enhances the maintainability and readability of the code.
- Applicable to various types of linked lists including singly, doubly, and circular linked lists.