Govt. Polytechnic Jhajjar

E-Contents

Name of the Faculty : **Harish Kumar Kaushik**

Discipline : **Computer Engg**

Semester : **IV**

Subject : **DSC**

Topic:

Introduction to Linked Lists

Introduction to Linked Lists

What is Linked List ?

A linked list is a fundamental data structure used in computer science to organize data. Unlike arrays, linked lists consist of nodes where each node contains two components:

1. Data : The actual value or data element.
2. Pointer (or Reference) : A reference to the next node in the sequence.

Importance of linked lists in Data Structures:

Dynamic Size: Unlike arrays, linked lists can grow or shrink in size dynamically, allowing efficient memory usage.

Ease of Insertion/Deletion: Inserting or deleting elements in a linked list is more efficient than in an array, as it only requires changing a few pointers rather than shifting elements.

Memory Utilization: Linked lists are suitable for applications where memory utilization is a concern, as they do not require a contiguous block of memory.

Flexibility: They provide flexibility in data organization, which is particularly useful for certain data structures like stacks, queues, and graphs.

Efficient Data Rearrangement: Linked lists allow efficient rearrangement of data elements, making them ideal for applications like implementing adjacency lists in graphs or creating hash tables with chaining for collision resolution.

Basic Concepts:

- **Node Structure:**

```
struct Node {
    int data;          // This will hold the data
    struct Node* next;  // This will point to the next node in the list
};
```

- **Creating a node:**

```
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

Types of Linked Lists:

1. Singly Linked List: In a singly linked list, each node contains a data field and a pointer to the next node in the sequence. The last node points to NULL.

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

Characteristics
Simple structure: Easy to implement.
Unidirectional traversal: Can only traverse in one direction (from the head to the end).

Example Operations
Insertion: Easy at the beginning or end; more complex in the middle.
Deletion: Requires finding the previous node to update its next pointer.

2. Doubly Linked List: In a doubly linked list, each node contains a data field, a pointer to the next node, and a pointer to the previous node.

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

Characteristics
Bidirectional traversal: Can traverse in both directions (forward and backward).
More complex: Requires handling two pointers in each node.

Example Operations
Insertion and deletion: Easier and more flexible since you can move in both directions.
Memory usage: Requires more memory due to the additional pointer.

3. Circular Linked List : In a circular linked list, the last node points back to the first node, forming a circle.

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

Characteristics
Circular nature: The next pointer of the last node points to the first node.
No null termination: There is no node with a NULL next pointer.

Example Operations
Traversal: Can start at any node and traverse the entire list.
Use cases: Useful for applications requiring a circular iteration, like round-robin scheduling.

4. Circular Doubly Linked List: A circular doubly linked list combines the features of a doubly linked list and a circular linked list. Each node points to both its next and previous nodes, and the last node points back to the first node.

```
struct Node {
    int data;
```

```
    struct Node* next;
    struct Node* prev;
};
```

Characteristics
Circular and bidirectional: Can traverse in both directions and form a circle.
Complex structure: More pointers to manage compared to singly or doubly linked lists.

Example Operations
Traversal: Can traverse the list starting from any node and move in either direction.
Insertion and deletion: More flexible but requires careful pointer management.

Operations on Linked Lists:

1. Insertion:

- At Beginning:

```
void insertAtBeginning(struct Node** head_ref, int new_data) {
    struct Node* new_node = createNode(new_data);
    new_node->next = *head_ref;
    *head_ref = new_node;
}
```

- At end:

```
void insertAtEnd(struct Node** head_ref, int new_data) {
    struct Node* new_node = createNode(new_data);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    struct Node* last = *head_ref;
```

```
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = new_node;
}
```

2. Deletion:

- From Beginning:

```
void deleteFromBeginning(struct Node** head_ref) {
    if (*head_ref == NULL) return;
    struct Node* temp = *head_ref;
    *head_ref = (*head_ref)->next;
    free(temp);
}
```

- From end:

```
void deleteFromEnd(struct Node** head_ref) {
    if (*head_ref == NULL) return;
    if ((*head_ref)->next == NULL) {
        free(*head_ref);
        *head_ref = NULL;
        return;
    }
    struct Node* temp = *head_ref;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}
```

3. Traversal:

```
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}
```

4. 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;
}
```

5. Reversing:

```
void reverseList(struct Node** head_ref) {
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}
```

Applications and Use Cases:

1. Implementing Stacks

Description: Stacks follow a Last In, First Out (LIFO) principle, where the last element added is the first to be removed.

Use Case: Linked lists can be used to implement stacks efficiently by performing insertions and deletions at the head of the list.

Example: Undo functionality in software applications, where the last action needs to be reversed first.

2. Implementing Queues

Description: Queues follow a First In, First Out (FIFO) principle, where the first element added is the first to be removed.

Use Case: Linked lists can be used to implement queues by inserting elements at the end and removing them from the beginning.

Example: Print queue management, where documents are printed in the order they were added.

3. Implementing Deques

Description: Double-ended queues (deques) allow insertion and deletion at both ends.

Use Case: Linked lists can be used to efficiently implement deques, supporting operations at both the head and tail.

Example: Task scheduling systems that require operations at both ends of the queue.

4. Adjacency Lists for Graphs

Description: Adjacency lists represent graphs by maintaining a list of adjacent vertices for each vertex.

Use Case: Linked lists are used to store adjacency lists, making it efficient to traverse and update edges.

Example: Social networks, where each user is represented as a node and connections (friendships) are edges.

5. Dynamic Memory Management

Description: Linked lists can be used to manage free memory blocks in dynamic memory allocation systems.

Use Case: Operating systems and memory allocators use linked lists to keep track of free and used memory blocks.

Example: Managing heap memory in programming languages with dynamic memory allocation (e.g., C's malloc and free).

6. Music Playlist Management

Description: Music players often use linked lists to manage playlists, allowing easy addition, removal, and reordering of songs.

Use Case: Linked lists provide flexibility to manipulate the playlist without significant overhead.

Example: Media players like Spotify or iTunes.

7. Undo Functionality in Text Editors

Description: Text editors use linked lists to store the sequence of changes made to a document, allowing users to undo and redo actions.

Use Case: Each state of the document is stored as a node, enabling easy traversal through the history of changes.

Example: Text editors like Microsoft Word or Sublime Text.

8. Polynomial Arithmetic

Description: Polynomials can be represented using linked lists where each node represents a term.

Use Case: Linked lists allow efficient implementation of polynomial addition, subtraction, and multiplication.

Example: Computer algebra systems.

9. Browser History Management
Description: Browsers use linked lists to maintain a history of visited web pages, allowing forward and backward navigation.
Use Case: Linked lists make it easy to add new entries and navigate through the history.
Example: Web browsers like Google Chrome or Mozilla Firefox.

10. Hash Tables with Chaining

Description: Linked lists are used to handle collisions in hash tables through chaining.

Use Case: Each bucket of the hash table is a linked list that stores all elements with the same hash index.

Example: Dictionaries in programming languages like Python or Java.
