# Chapter 3 : Linked Lists

## 3.1 Introduction

Efficient data organization is a central concern in computer science. One of the simplest structures is the array, where elements are stored in contiguous memory locations. Arrays provide fast access using indices, but they come with important limitations: their size is fixed, and inserting or deleting elements can be costly because other elements must be shifted.

To address these limitations, linked lists are used. A linked list is a dynamic data structure composed of nodes that are connected through pointers. Each node contains a value and a reference to the next node. Because nodes are allocated independently in memory, the structure can grow or shrink during execution, making it more flexible than arrays.

## 3.2 Pointers

Understanding pointers is essential, as linked lists rely entirely on them.

### 3.2.1 What is a Pointer?

A pointer is a variable that stores the memory address of another variable rather than a direct value.

```
int x = 10;
int *p = &x;
```

Here, `x` stores the value `10`, while `p` stores the address of `x`. The symbol `&` is used to obtain the address of a variable.

### 3.2.2 Dereferencing a Pointer

Accessing the value stored at a given address is called **dereferencing**.

```
printf("%d", *p);
```

The expression `*p` retrieves the value stored at the address contained in `p`. In this case, it prints `10`.

### 3.2.3 Pointers and Memory

Each variable occupies a specific location in memory. Pointers provide direct access to these locations, which allows:

- Efficient data manipulation
- Sharing data between functions
- Construction of dynamic structures

Because pointers operate at a low level, careful handling is required to avoid errors.

### 3.2.4 Dynamic Memory Allocation

In many situations, the amount of memory needed is not known in advance. C allows allocating memory during program execution using functions from **<stdlib.h>**. This mechanism is called dynamic memory allocation and is fundamental for linked lists.

### 3.2.5 The malloc Function

The `malloc` function (memory allocation) reserves a block of memory and returns a pointer to it.

```
pointer = (type*) malloc(size);
```

- `size` represents the number of bytes to allocate.
- The returned value is a pointer to the allocated memory.
- If allocation fails, NULL is returned.

**Example:**

```
int *p = (int*) malloc(sizeof(int));
```

This allocates enough memory to store one integer. The use of `sizeof(int)` ensures portability across systems.

Once allocated, the memory can be used normally:

```
*p = 25;
```

### 3.2.6 Allocating Multiple Elements

Dynamic allocation can also be used to store several elements:

```
int *arr = (int*) malloc(5 * sizeof(int));
```

This creates space for five integers, which can be accessed like an array.

### 3.2.7 The free Function

Memory allocated with `malloc` must be released when it is no longer needed:

```
free(pointer);
```

Releasing memory prevents waste and keeps the program efficient.

### 3.2.8 Memory Leaks

Failing to release memory leads to **memory leaks**, where unused memory remains allocated. Over time, this can slow down or even crash a program.

### 3.2.9 Good Practices

Safe use of pointers includes:

- Checking if allocation was successful
- Freeing memory after use
- Avoiding access to freed memory

**Example :**

```
int *p = (int*) malloc(sizeof(int));
if (p == NULL) {
    printf("Allocation failed\n");
    return 1;
}
*p = 10;
free(p);
p = NULL;
```

Setting the pointer to `NULL` after freeing it helps prevent accidental misuse.

## 3.3 Linked Lists

A linked list is built by chaining nodes together using pointers.

### 3.3.1 Node Definition Using typedef

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

This definition creates a new type `Node`. Each node stores an integer and a pointer to the next node. The use of `typedef` simplifies the syntax by avoiding repeated use of `struct`.

### 3.3.2 Creating a Node

Nodes are created dynamically using `malloc`.

```
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));

    if (newNode == NULL) {
        printf("Allocation failed\n");
        return NULL;
```

```
    }

    newNode->data = value;
    newNode->next = NULL;

    return newNode;
}
```

The new node is initialized with a value and a `NULL` pointer, indicating that it does not yet point to another node.

### 3.3.3 Representation

A linked list can be visualized as a sequence of nodes:

```
10 -> 20 -> 30 -> NULL
```

Each element points to the next one, and the last node marks the end with NULL.

## 3.4 Operations on Linked Lists

Several basic operations are performed on linked lists.

### 3.4.1 Traversal

Traversal consists of visiting each node in sequence.

```
void printList(Node* head) {
    Node* temp = head;

    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }

    printf("NULL\n");
}
```

The process starts at the `head` and continues until reaching `NULL`.

### 3.4.2 Insertion

Adding elements can be done in different positions.

**Insert at Beginning**

```
Node* insertAtBeginning(Node* head, int value) {
    Node* newNode = createNode(value);

    if (newNode == NULL)
        return head;

    newNode->next = head;
    return newNode;
}
```

The new node becomes the first element, pointing to the previous head.

**Insert at End**

```
Node* insertAtEnd(Node* head, int value) {
    Node* newNode = createNode(value);

    if (newNode == NULL)
        return head;

    if (head == NULL)
        return newNode;

  Node* temp = head;

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

    temp->next = newNode;

    return head;
}
```

The list is traversed until the last node, then the new node is attached.

### 3.4.3 Deletion

Removing elements must also free the associated memory.

```
Node* deleteFirst(Node* head) {
    if (head == NULL)
        return NULL;

    Node* temp = head;
    head = head->next;

    free(temp);

    return head;
}
```

The first node is removed and its memory is released.

### 3.4.4 Searching

Searching consists of finding a value in the list.

```
int search(Node* head, int value) {
    Node* temp = head;

    while (temp != NULL) {
        if (temp->data == value)
            return 1;
        temp = temp->next;
    }

    return 0;
}
```

Each node is examined until the value is found or the end is reached.

## 3.5 Doubly Linked Lists

A doubly linked list extends the idea by linking nodes in both directions.

### 3.5.1 Node Definition

```
typedef struct DNode {
    int data;
    struct DNode* prev;
    struct DNode* next;
} DNode;
```

Each node contains pointers to both the previous and next nodes.

### 3.5.2 Creating a Node

```
DNode* createDNode(int value) {
    DNode* newNode = (DNode*)malloc(sizeof(DNode));

    if (newNode == NULL) {
        printf("Allocation failed\n");
        return NULL;
    }

    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;

    return newNode;
}
```

### 3.5.3 Insertion at Beginning

```
DNode* insertBeginning(DNode* head, int value) {
    DNode* newNode = createDNode(value);

    if (newNode == NULL)
        return head;
```

```
    if (head != NULL) {
        head->prev = newNode;
        newNode->next = head;
    }

    return newNode;
}
```

Both forward and backward links must be updated.

### 3.5.4 Advantages and Disadvantages

Doubly linked lists allow traversal in both directions and simplify certain operations like deletion. However, they require more memory and involve more complex pointer management.

## 3.6 Conclusion

Linked lists provide a flexible alternative to arrays by using pointers and dynamic memory allocation. Mastering concepts such as `malloc`, `free`, and pointer manipulation is essential for working with these structures and for progressing toward more advanced topics in computer science.