# Linked Lists

## A Definition

A linked list is a linear data structure where each element, called a node, contains a value and a reference (or pointer) to the next node in the sequence. This structure allows for efficient insertion and deletion of elements, as well as dynamic resizing of the list. They are particularly useful when the size of the data structure is not known in advance or when frequent insertions and deletions are required. Applications of linked lists include:

- Implementing stacks and queues
- Representing graphs and trees
- Managing memory in dynamic data structures
- Implementing hash tables with separate chaining
- Maintaining a list of items that can grow and shrink in size
- Implementing adjacency lists for graphs
- Creating undo/redo functionality in applications
- Implementing sparse matrices
- Building linked data structures like skip lists
- Implementing LRU (Least Recently Used) caches
- Creating circular linked lists for round-robin scheduling
- Implementing polynomial arithmetic

## B Types of Linked Lists

### B.1 Singly Linked List

A singly linked list is a type of linked list where each node contains a value and a pointer to the next node in the sequence. The last node points to null, indicating the end of the list. This structure allows for efficient insertion and deletion at both ends of the list. They are particularly useful for applications where the size of the data structure is not known in advance or when frequent insertions and deletions are required.

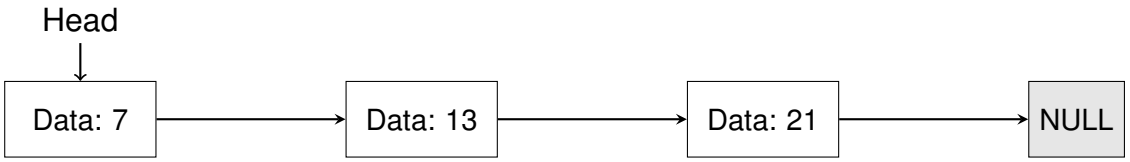


Figure 1: A simple singly linked list with three nodes.

### B.2 Doubly Linked List

A doubly linked list is a type of linked list where each node contains a value, a pointer to the next node, and a pointer to the previous node. This allows for traversal in both directions, making it easier to insert and delete nodes from both ends of the list. They find applications in scenarios where bidirectional traversal is required, such as in navigation systems or undo/redo functionality in applications.

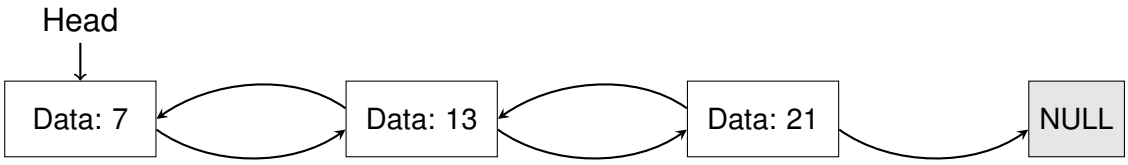


Figure 2: A simple doubly linked list with three nodes.

### B.3 Circular Linked List

A circular linked list is a type of linked list where the last node points back to the first node, creating a circular structure. This allows for continuous traversal of the list without reaching a null pointer. Circular linked lists can be singly or doubly linked. They are used in applications where the list needs to be traversed repeatedly, such as in round-robin scheduling or in implementing circular buffers.

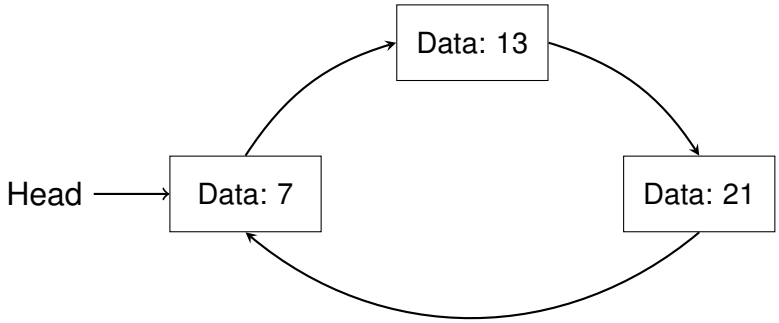


Figure 3: A simple circular singly linked list with three nodes.

## C Operations on Linked Lists

### C.1 Insertion

Insertion in a linked list involves adding a new node at a specific position. The operation can be performed at the beginning, end, or any position in the list. The steps for insertion are as follows:

1. Create a new node with the desired value.
2. If inserting at the beginning, update the new node's next pointer to point to the current head and set the head to the new node.
3. If inserting at the end, traverse to the last node and update its next pointer to point to the new node.
4. If inserting at a specific position, traverse to the node before the desired position, update its next pointer to point to the new node, and set the new node's next pointer to point to the next node in the list.
5. If the list is empty, set the head to the new node.

Figure 4 illustrates the insertion of a new node with value 10 between nodes with values 13 and 21. The pointer from node 13 is updated to point to the new node, and the new node's next pointer is set to point to node 21.

### C.2 Deletion

Deletion in a linked list involves removing a node from a specific position. The steps for deletion are as follows:

1. If the list is empty, return.

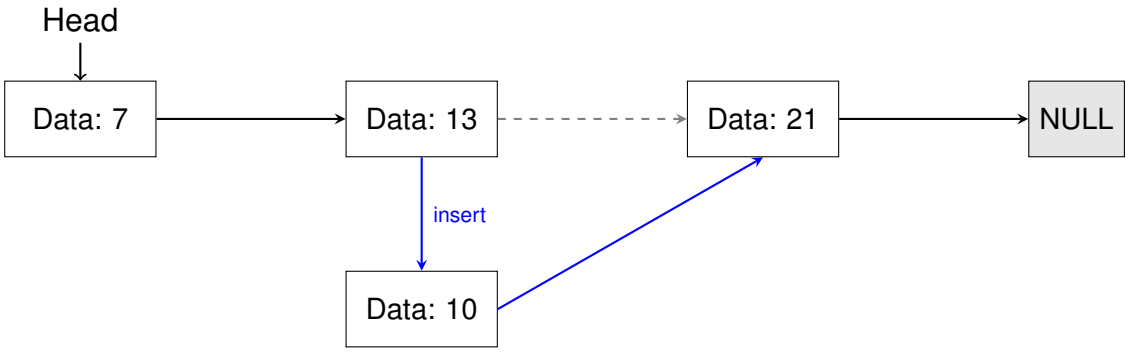


Figure 4: Insertion of a new node in a singly linked list

2. If deleting the head node, update the head to point to the next node.
3. If deleting a node at a specific position, traverse to the node before the desired position, update its next pointer to skip the node to be deleted, and set it to point to the next node in the list.
4. If deleting the last node, traverse to the second-to-last node and set its next pointer to null.
5. If the node to be deleted is not found, return without making any changes.

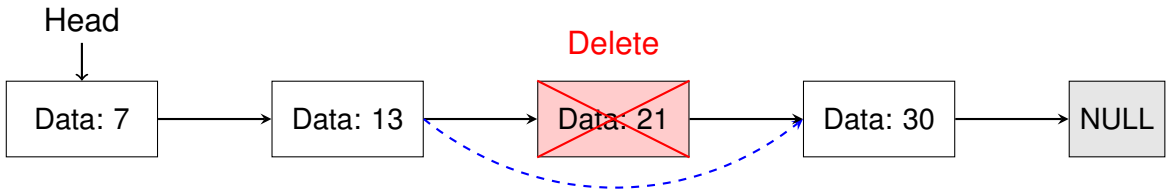


Figure 5: Deletion of a node in a singly linked list

Figure 5 illustrates the deletion of the node with value 21. The pointer from node 13 is updated to point to node 30, effectively bypassing the deleted node.

### C.3 Traversal

Traversal, as shown in Figure 6, in a linked list involves visiting each node in the list to access or display its value. The steps for traversal are as follows:

1. Start at the head of the list.
2. While the current node is not null, perform the desired operation (e.g., print the value).
3. Move to the next node by updating the current node to its next pointer.
4. Repeat until the end of the list is reached (i.e., when the current node is null).
5. If the list is empty, return without performing any operations.

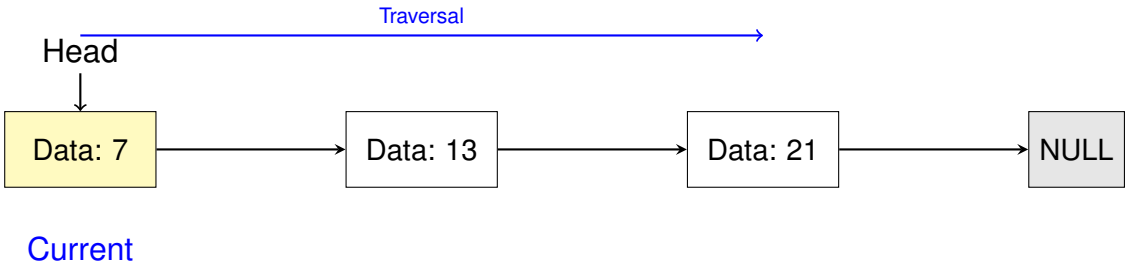


Figure 6: Traversal of a singly linked list: The highlighted node indicates the current node during traversal.

**Practice Exercise:**

1. Implement a doubly linked list in your preferred programming language. Each node should contain:
   - A value (e.g., integer)
   - A pointer/reference to the next node
   - A pointer/reference to the previous node
2. Implement the following functionalities:
   - **Insert** a new node at the beginning, end, and at a specific position
   - **Delete** a node from the beginning, end, and a specific position
   - **Traverse** the list forward and backward, printing the values of each node
   - **Search** for a value in the list and return its position(s)
3. Write test cases to demonstrate each operation.
4. (Optional) Implement a function to reverse the doubly linked list.

## D Pointer Base Representation

A linked list is typically represented using pointers, where each node contains a value and a pointer to the next node. The head of the list points to the first node, and the last node's pointer points to null (or None in Python). This representation allows for dynamic memory allocation and efficient insertions and deletions.

**Example: C++ Example of a Doubly Linked List**

```cpp
#include <iostream>

struct Node {
    int data;
    Node* next;
    Node* prev;
    Node(int val) : data(val), next(nullptr), prev(nullptr) {}
};

class DoublyLinkedList {
public:
    Node* head;
    Node* tail;
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    void push_front    "latex-workshop.latex.outDir":
       "./build"(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
    }

    void printForward() {
        Node* curr = head;
        while (curr) {
            std::cout << curr->data << "␣";
            curr = curr->next;
        }
        std::cout << std::endl;
    }

    void printBackward() {
        Node* curr = tail;
        while (curr) {
            std::cout << curr->data << "␣";
            curr = curr->prev;
        }
        std::cout << std::endl;
    }
};
```

## E Advantages and Disadvantages

### E.1 Advantages

- Dynamic Size: Linked lists can grow and shrink in size as needed, unlike arrays which have a fixed size.
- Efficient Insertions/Deletions: Insertion and deletion operations can be performed in

constant time if the position is known, as they do not require shifting elements like in arrays.

- Memory Utilization: Linked lists can utilize memory more efficiently, as they do not require contiguous memory allocation.
- Flexibility: They can easily implement complex data structures like stacks, queues, and graphs.
- No Wasted Space: Unlike arrays, linked lists do not have unused space when elements are removed, as they can dynamically adjust their size.

### E.2 Disadvantages

- Memory Overhead: Each node requires additional memory for storing pointers, which can lead to higher memory usage compared to arrays.
- Sequential Access: Linked lists do not allow random access to elements, making it slower to access elements compared to arrays.
- Cache Locality: Linked lists may have poor cache performance due to non-contiguous memory allocation, leading to more cache misses.
- Complexity: Implementing linked lists can be more complex than using arrays, especially when dealing with pointer manipulation.
- Difficulty in Reversing: Reversing a linked list is more complex than reversing an array, as it requires changing pointers rather than simply swapping elements.
- Increased Complexity: The implementation of linked lists can be more complex than arrays, especially when dealing with pointer manipulation and memory management.
- No Direct Access: Unlike arrays, linked lists do not allow direct access to elements, which can lead to slower access times for certain operations.
- Fragmented Memory: Linked lists may lead to fragmented memory allocation, as nodes can be scattered throughout memory rather than being contiguous.