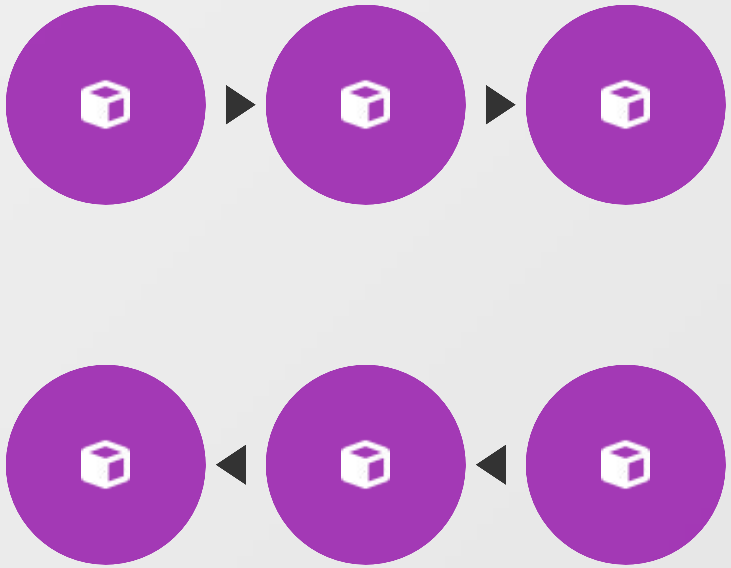
Data Structures

A Deep Dive into Doubly Linked Lists

Comprehensive Exploration of Concepts, Implementations, and Applications



Course: **Data Structures & Algorithms**

Presented by: **Dr. Bahadir Aydin**

Report Generated: 2025-09-03

Course Agenda

Definition and Concepts
Understanding the fundamental principles of doubly linked lists

Structure and Nodes
Examining the anatomy of nodes in doubly linked lists

Core Operations
Traversal, insertion, and deletion techniques

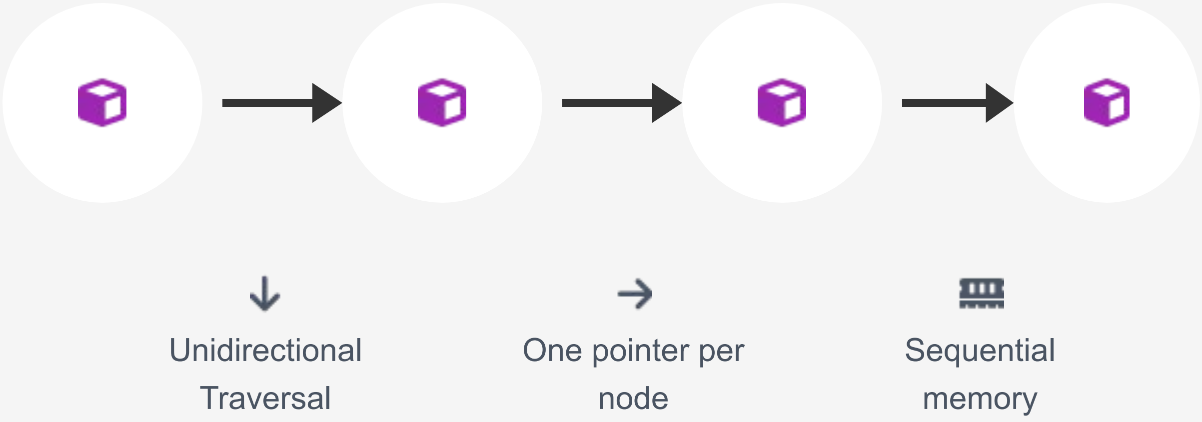
Implementation
Examples in programming languages with practical demonstrations

Complexity Analysis
Time and space efficiency of doubly linked list operations

Pros and Cons
Advantages and disadvantages compared to other data structures

Real-World Applications
Practical use cases in computing and software development

Recap: Singly Linked List



Key Characteristics:

Linear Structure
Each node contains data and a reference to the next node

One-way Traversal
Can only be traversed from head to tail

Sequential Access
Must start from beginning to access elements

Deletion Limitation
Requires traversal to find previous node

What is a Doubly Linked List?

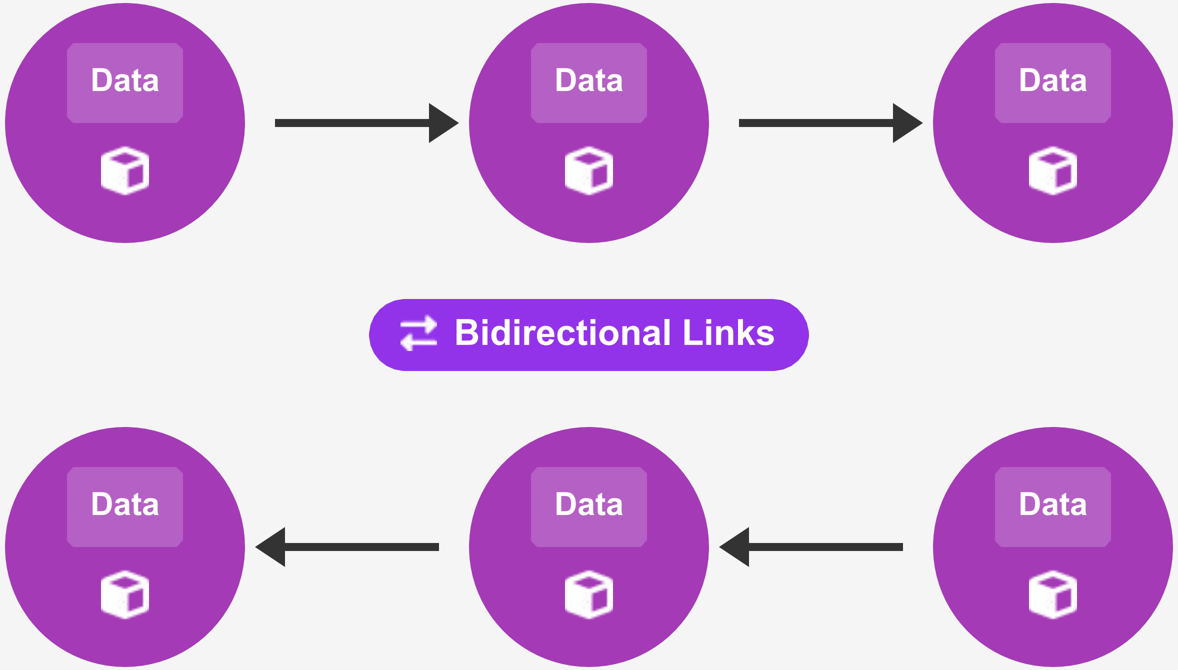
Doubly Linked List (DLL)

A more advanced type of linked list where each node contains:

- The actual **data** value
- A pointer to the **next** node
- A pointer to the **previous** node

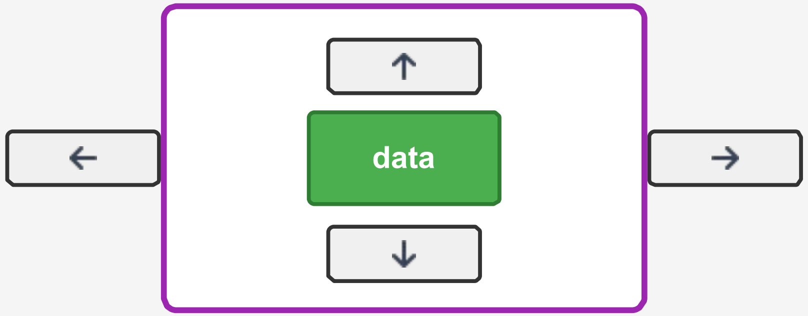
Key Advantage

Enables **bidirectional traversal** — forward → and backward ← through the list



Compared to singly linked lists, DLLs offer greater operational flexibility at the cost of increased memory usage.

Anatomy of a Node




```
class Node {
public:
    // To store the Value or data
    int data;
    // Pointer to point the Previous Element
    Node* prev;
    // Pointer to point the Next Element
    Node* next;
```

Data Field

- Stores the actual value or information
- Can be of any data type (int, string, object)
- Represents the element stored in the list

Next Pointer

- Points to the next node in sequence
- If last node, typically NULL or nullptr
- Enables forward traversal

Previous Pointer

- Points to the previous node
- If first node (head), typically NULL or nullptr
- Enables backward traversal

Singly vs. Doubly Linked List

Singly Linked List Node

Data + **next** pointer

Doubly Linked List Node

Data + **next** + **prev** pointer

Attribute	Singly Linked List	Doubly Linked List
Node Structure	Data, single `next`	Data, *`prev`*, *`next`*

Traversal Operations

Doubly Linked Lists allow bidirectional traversal thanks to the `next`

Insertion: At the Beginning

Step-by-Step Process

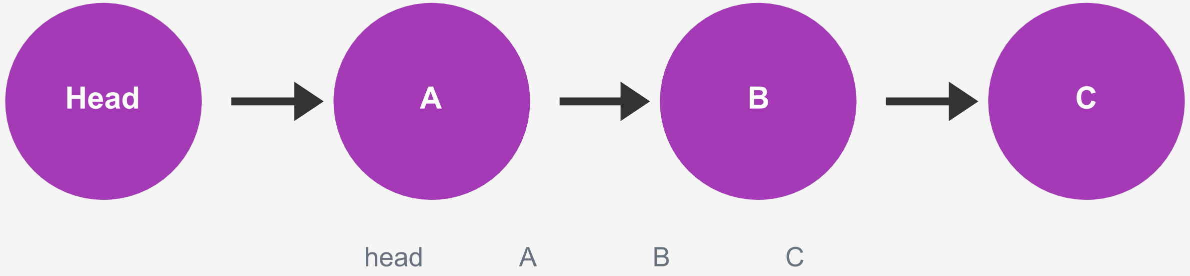
1. **Create New Node**
   Allocate memory for the new node and assign the data value
2. **Set Next Pointer**
   Point newNode.next to the current head of the list
3. **Set Previous Pointer**
   Set newNode.prev to NULL (as it will be the new head)
4. **Update Current Head**
   If list is not empty, set head.prev to newNode
5. **Update Head Reference**
   Move the head pointer to newNode

Edge Case

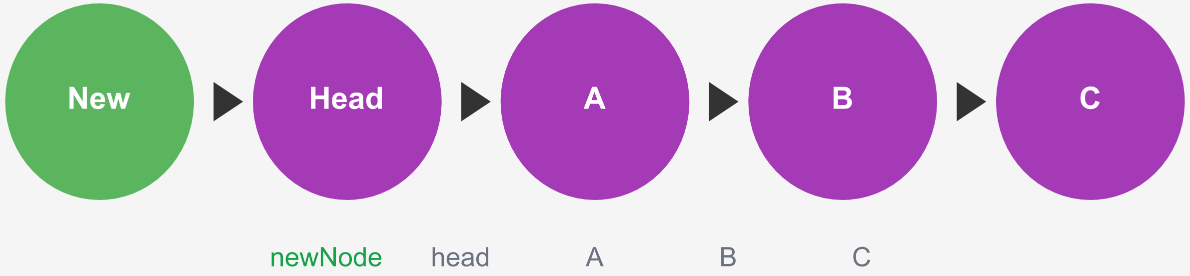
If the list is empty, both head and tail need to be updated to point to the new node

Visual Representation

Before Insertion



After Insertion



Key Pointer Updates

- newNode.next = head
- newNode.prev = NULL
- head.prev = newNode
- head = newNode

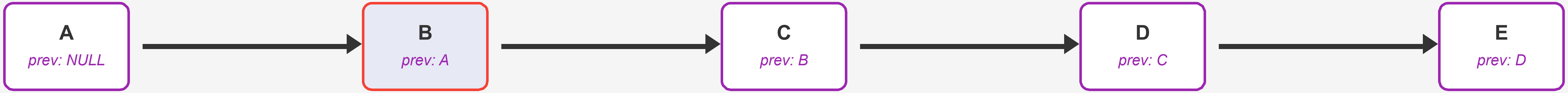
Insertion: At the End

Without Tail Pointer

1. Create new node with data
2. Set `newNode.next = NULL`

Insertion: At a Specific Position

Before Insertion



After Insertion

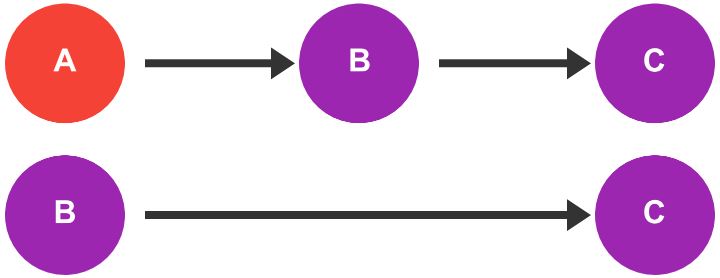
Insertion Algorithm

1. Create a new node (**newNode**) with the desired data
2. Set **newNode.next** to **prev_node.next**
3. Set **newNode.prev** to **prev_node**
4. Set **prev_node.next** to **newNode**
5. If **newNode.next** is not NULL (i.e., not inserting at the end), set **newNode.next.prev** to **newNode**

Key Insight: Inserting at a specific position requires updating **four pointers** to maintain bidirectional links.

Deletion Operations

From Beginning

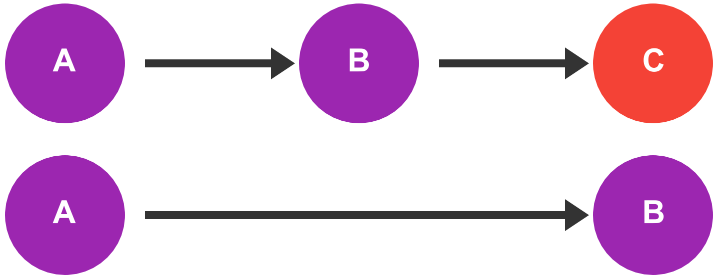


Key Steps:

1. Store head in temp
2. Update head to head.next
3. Set head.prev to NULL
4. Deallocate temp

Time Complexity: O(1)

From End

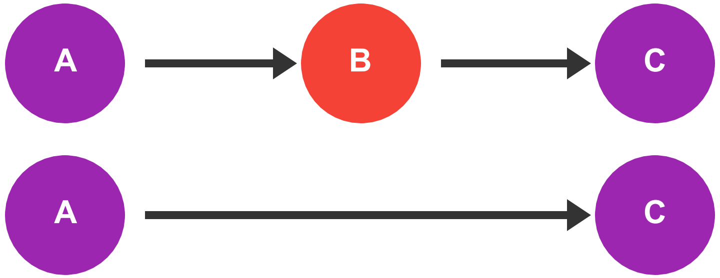


Key Steps:

1. Find second-to-last node
2. Set secondLast.next to NULL
3. Deallocate last node

Time Complexity: O(n)

At Position



Key Steps:

1. Update prev node's next
2. Update next node's prev
3. Bypass the node
4. Deallocate the node

Time Complexity: O(n)

Implementation: Node and List Classes

Node Class

```
class Node:
  def __init__(self, data):
    # To store the value or data.
    self.data = data
    # Reference to the previous node
    self.prev = None
    # Reference to the next node
    self.next = None
```

List Class

```
class DoublyLinkedList:
  def __init__(self):
    # Reference to the first node
    self.head = None
    # Reference to the last node (optional)
    self.tail = None
```

Node Class

- Encapsulates **data** and two pointers
- **prev**: Points to previous node
- **next**: Points to next node

List Class

- Manages the linked nodes
- **head**: First node in the list
- **tail**: Last node (optional)

Key Insight: The `tail`

Code Example: Insertion Implementation

```python
# Insert a new node at the end of the list
def insert_end(self, data):
    # Allocate memory for newNode and assign data
    new_node = Node(data)

    # If list is empty, make newNode head and tail
    if self.head is None:
        self.head = new_node
        self.tail = new_node
        return

    # Link current tail's next to new node
    self.tail.next = new_node
    # Link new node's prev to current tail
    new_node.prev = self.tail

    # Update tail to be the new node
    self.tail = new_node
```

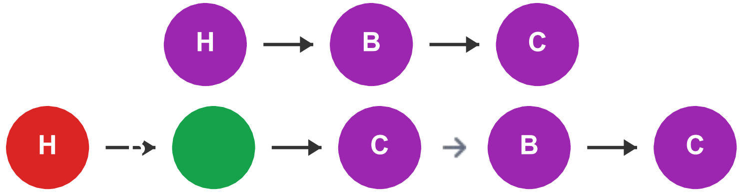
Code Example: Deletion Implementation

```
# Deleting a node from a Doubly Linked List

class DoublyLinkedList:
    # ... (previous Node and DoublyLinkedList code) ...

    def delete_node(self, del_node):
        # If head or del_node is None, deletion is not possibleif
        self.head is None or del_node is None:
            return

        # If del_node is the head node, update the head pointerif
        self.head == del_node:
            self.head = del_node.next

        # If del_node is not the last node, update the prev pointer of
        the node after del_node
        if del_node.next is not None:
            del_node.next.prev = del_node.prev
        else: # If del_node is the tail node, update the tail pointer
            self.tail = del_node.prev
        # If del_node is not the first node, update the next pointer of
        the node before del_node
        if del_node.prev is not None:
            del_node.prev.next = del_node.next
# The node is now unlinked and can be garbage collected (in Python)
# In languages like C/C++, explicit memory deallocation would be needed
here
```

Deletion Scenarios

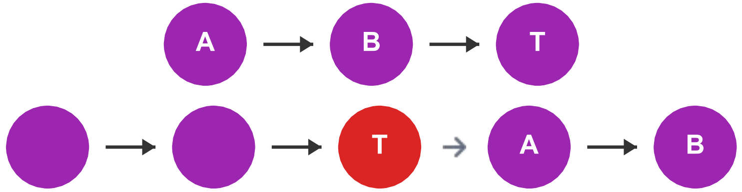
The `delete_node` function handles three main scenarios:

Deleting the Head Node



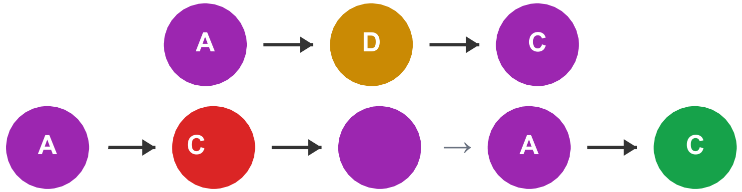
Update `head` to `head.next`

Deleting the Tail Node



Update tail to `del_node.prev`

Deleting the Intermediate Node



Update `del_node.prev.next` to `del_node.next`

Complexity Analysis

The efficiency of Doubly Linked List operations is typically analyzed using Big O notation, which describes the performance or complexity of an algorithm.

Operation	Time Complexity	Space Complexity	Description
Access (by index)	O(n)	O(1)	Requires traversal from head to find element
Search (by value)	O(n)	O(1)	Sequential search through list
Insertion (at beginning)	O(1)	O(1)	Direct access to head pointer
Insertion (at end)	O(1)	O(1)	With tail pointer, direct access to end
Insertion (at specific position)	O(n)	O(1)	Requires traversal to position
Deletion (from beginning)	O(1)	O(1)	Direct access to head pointer
Deletion (from end)	O(1)	O(1)	With tail pointer, direct access to end
Deletion (at specific position)	O(n)	O(1)	Requires traversal to position

Note: Insertion and deletion at a specific position require traversal to that position, leading to O(n) time complexity. However, if a pointer to the node to be deleted is already available, deletion can be O(1).

O(n) Linear time   O(1) Constant time

Advantages of Doubly Linked Lists

Bidirectional Traversal

Allows traversal in both forward and backward directions, enabling more flexible navigation through the list.

Efficient Deletion

Given a node pointer, deletion can be performed in O(1) time since the previous node is directly accessible.

Efficient Operations at Both Ends

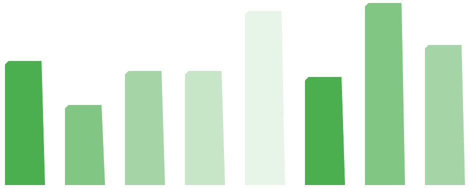
Insertion and deletion at the head or tail of the list are highly efficient, typically taking O(1) time.

Easier Implementation of Deque Features

Bidirectional nature simplifies implementation of data structures like Deques and navigation features.

Dynamic Size Adjustment

The list can easily grow or shrink as elements are added or removed, offering flexibility in memory allocation.



Key Insight: The bidirectional nature of doubly linked lists provides greater flexibility compared to singly linked lists, particularly when bidirectional traversal or efficient deletion is required.

Disadvantages of Doubly Linked Lists

Increased Memory Consumption

Each node requires an additional `prev`

Real-World Applications

Doubly Linked Lists provide efficient solutions for various real-world problems:

Web Browser Navigation

Browser history management. "Back" and "forward" buttons use DLLs to track visited pages, enabling bidirectional navigation.

Undo/Redo Functionality

Text editors and design tools use DLLs to maintain history of actions, enabling efficient revert or reapply changes.

Media Playlists

Media players use DLLs to manage song/video playlists, allowing seamless movement between tracks in both directions.

MRU/LRU Caches

DLLs are fundamental in building caches that efficiently track and manage recently accessed items.

Thread Schedulers in Operating Systems

Operating systems use DLLs to manage processes or threads, allowing schedulers to efficiently move between active tasks.

Key Insight: Doubly Linked Lists are ideal for applications requiring bidirectional navigation or efficient element manipulation.

Implementation Considerations

Key considerations when implementing doubly linked lists:

Memory Management

- Proper allocation for new nodes
- Deallocation when deleting nodes
- Prevent memory leaks

Pointer Handling

- Update `next`

Summary and Key Takeaways

Doubly Linked Lists are versatile data structures with bidirectional traversal capabilities

Versatility

Bidirectional traversal and efficient operations at both ends

Trade-offs

Increased memory vs. simplified operations

Applications

Browser history, undo/redo, media players

Final Thoughts

- DLLs provide **efficient deletion** when node pointer is known
- **Bidirectional traversal** enhances flexibility
- **Increased memory** due to additional pointers
- **Complex implementation** with two pointers

"Doubly Linked Lists offer enhanced capabilities with trade-offs in complexity"