DATA STRUCTURES CLASS

Linked Lists

A Comprehensive Overview

Exploring fundamental concepts, types, operations, advantages, disadvantages, and real-world applications of linked lists

Course: **Data Structures & Algorithms**

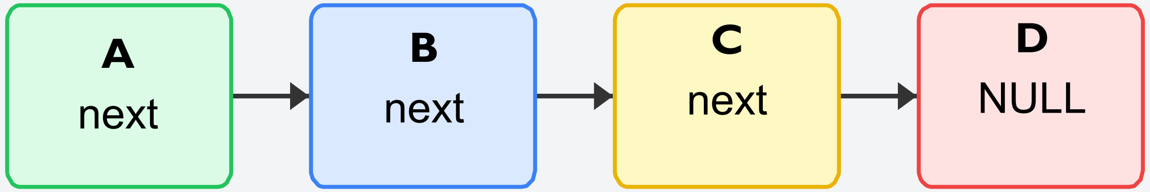
Presented by: **Dr. Bahadir Aydin**

What is a Linked List?

A linked list is a linear data structure where elements are not stored at contiguous memory locations.

Instead, each element (node) contains:

- **Data Field:** Stores the actual value
- **Pointer/Next Field:** Stores reference to the next node



NULL (or None) indicates the end of the list

Key Characteristics:

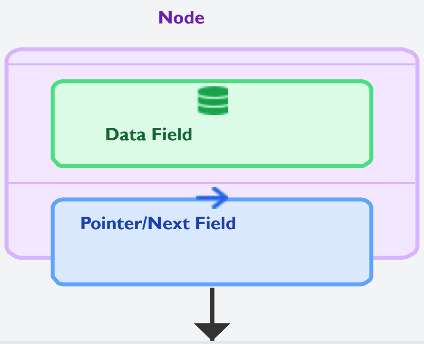
- Non-contiguous memory allocation
- Dynamic in size (unlike arrays with fixed size)
- Starting point identified by "HEAD" pointer
- End of list indicated by "NULL" pointer

Contrast with Arrays:

Arrays:
Contiguous memory allocation Fixed size

Linked Lists:
Non-contiguous memory Dynamic size

Components of a Linked List Node



Links to the next node in the sequence

Implementation Examples

Python

```python
class Node:
    def __init__(self, data):

        # Data part of the node
        self.data = data
        # Pointer to the next node
        self.next = None
```

C++

```cpp
struct Node {
    // Data part of the node
    int data;
    // Pointer to the next node
    Node* next;
    };
```

Key Points:

- **Data Field:** Can store any data type (integers, strings, objects)
- **Pointer Field:** Stores memory address of next node or NULL if it's the last node

Linked Lists vs. Arrays

Aspect	Linked Lists	Arrays
Memory Allocation	Non-contiguous; elements allocated individually	Contiguous; allocated as a single block
Element Access	Sequential; requires traversal from beginning	Random; direct access via index
Insertion/Deletion	Efficient (O(1)) if position is known	Inefficient (O(n)) for middle operations
Size	Dynamic; grows/shrinks during runtime	Fixed; size declared beforehand

Performance Analysis

Time Complexity Comparison

Operation	Linked Lists	Arrays
Access by Index	O(n)	O(1)
Search	O(n)	O(n)
Insertion at Beginning	O(1)	O(n)
Insertion at End	O(n)	O(1)

When to Use Each Structure

Linked Lists

- Dynamic data with frequent insertions/deletions
- Unknown number of elements

Arrays

- Fixed-size collections
- Frequent random access

Performance Analysis

Efficiency of operations on linked lists vs. arrays (n = number of elements):

	Linked List	Array
Access by Index	O(n)	O(1)
Search	O(n)	O(n)
Insertion at Beginning	O(1)	O(n)
Insertion at End	O(n)	O(1)
Deletion at Beginning	O(1)	O(n)

Linked List Advantages:

- Insert/Delete at Beginning: O(1) since only head pointer needs updating

Array Advantages:

- Access by Index: O(1) due to contiguous memory allocation

Space Efficiency Considerations

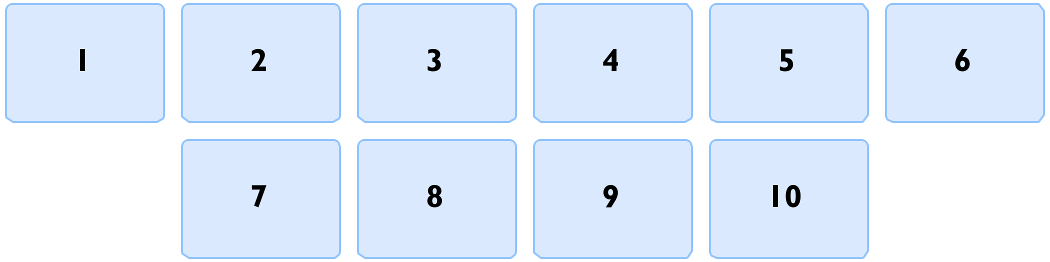
Memory Usage Comparison

Arrays

Memory Formula:
`n × size_of_data_element`

Example:

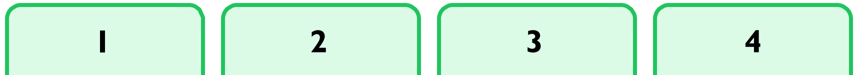
`10 integers = 10×4 = 40 bytes`



Singly Linked Lists

Memory Formula:
`n ×(size_of_data_element + size_of_pointer)`

Example:
`10 integers = 10×(4 + 8) = 120 bytes`

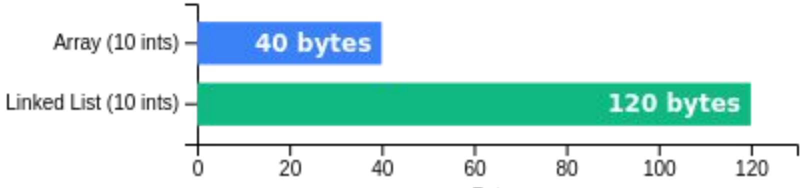


Space Efficiency Analysis

- Arrays are more memory-efficient for static collections with known size
- Linked lists introduce **pointer overhead** (typically 4-8 bytes per node)
- Memory overhead becomes more significant for smaller data types

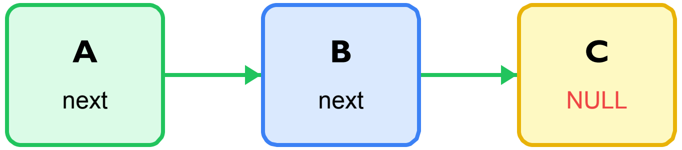
Dynamic Allocation Benefits

- Linked lists avoid **wasted memory** from oversized arrays
- They prevent **expensive reallocations** needed for dynamic arrays
- Memory is only allocated when needed, keeping total allocation optimized



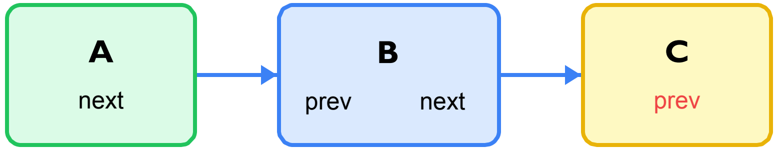
Types of Linked Lists

Singly Linked List



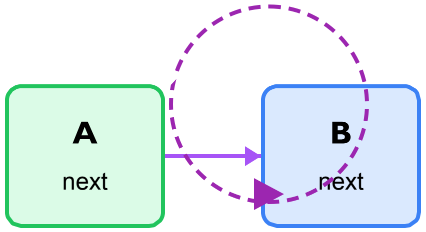
- Each node contains data and a pointer to the next node
- Traversal is unidirectional (from head to tail)
- Memory-efficient as each node only stores one pointer

Doubly Linked List



- Each node contains data, next pointer, and previous pointer
- Allows bidirectional traversal (forward and backward)
- More memory overhead but improves efficiency for certain operations

Circular Linked List



- Last node's next pointer points back to the first node
- No NULL pointer to signify end of list
- Useful for continuous cycling through elements

Each type has its own advantages and is suitable for different use cases

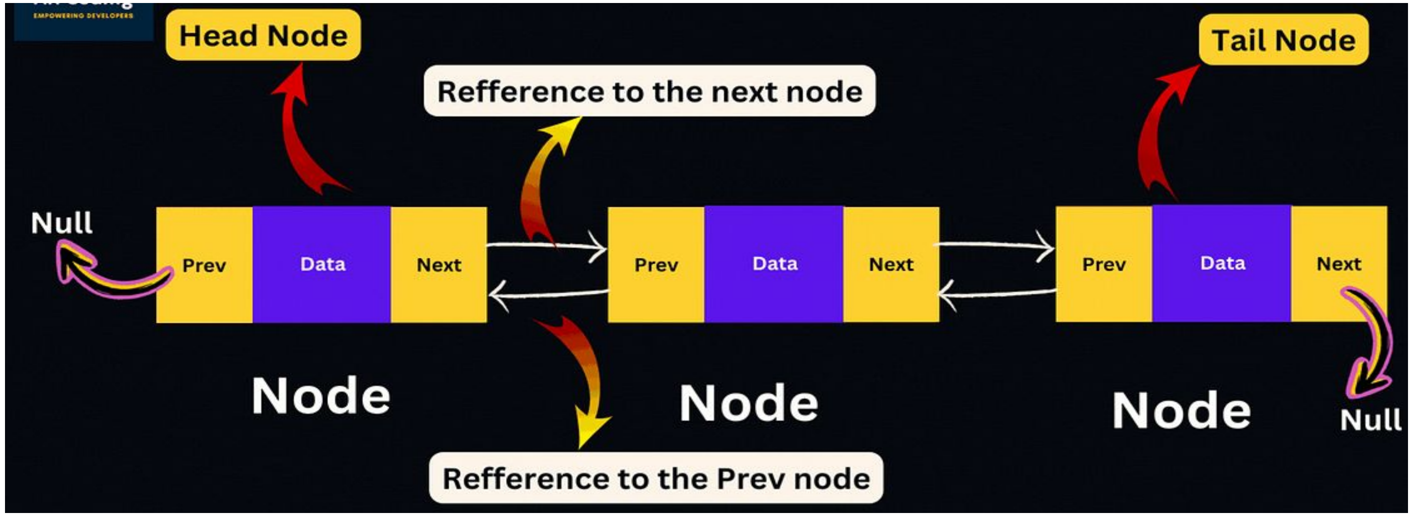
Singly Linked List Implementation

```
class Node:
    def   init (self, data):
        self.data = data  # Data part of the node
        self.next = None  # Pointer to the next node, initialized to None

class LinkedList:
    def init   (self):
        self.head = None  # Head of the linked list, initially None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node# If list is empty, new node becomes new
            return              # head
        current = self.head
        while current.next:
            current = current.next    # Traverse to the last
        node current.next = new_node  # Link the new node to
        the end
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")  # Print current node's data
            current = current.next    # Move to the next node
        print("None")
```

```
# Example Usage:
my_list = LinkedList()
my_list.append(1)
my_list.append(2)
my_list.append(3)
print("Singly Linked List:")
my_list.display()   # Output: 1 -> 2 -> 3 ->
None
```

Doubly Linked List Implementation

Doubly Linked List Structure:



Key Features:

- Each node has **three** components: data, next pointer, and previous pointer
- **Bidirectional** traversal capability (forward and backward)
- Additional **"prev"** pointer allows for efficient operations
- Maintains **"head"** and **"tail"** pointers for efficient list management

Python Implementation

```python
class Node:
    def _init_(self, data):
        self.data = data  # Data part of the node
        self.next = None  # Pointer to the next node
        self.prev = None  # Pointer to the previous node

class DoublyLinkedList:
    def _init_(self):
        self.head = None  # Head of the list
        self.tail = None  # Tail of the list (useful for O(1)

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            self.tail = new_node
            return
        self.tail.next = new_node  # Link current tail to new hd
        new_node.prev = self.tail  # Link new node back to cur
        self.tail = new_node     # Update tail to new
        node

    def display_forward(self):
        current = self.head
        print("Forward Traversal:")
        while current:
            print(current.data, end=" <-> ")
            current = current.next
        print("None")
```

```python
    def display_backward(self): current =
        self.tail
        print("Backward Traversal:") while
        current:
            print(current.data, end=" <-> ") current =
            current.prev
        print("None")

# Example Usage:
my_dll = DoublyLinkedList()
my_dll.append(1) my_dll.append(2)
my_dll.append(3)
my_dll.display_forward()     # Output: Forward Traversal: 1 <-> 2
my_dll.display_backward() # Output: Backward Traversal: 3 <->
```

Output:

Forward Traversal: 1 <-> 2 <-> 3 <-> None

Backward Traversal: 3 <-> 2 <-> 1 <-> None

Circular Linked List Implementation

```python
# compact circular singly linked list
class Node:
    def __init__(self, d, nxt=None): self.data, self.next = d, nxt
class CircularLinkedList:
    def __init__(self): self.head = None
    def is_empty(self): return self.head is None

    def traverse(self):
        if not self.head: return []
        out, cur = [], self.head
        while True:
            out.append(cur.data); cur = cur.next
            if cur is self.head: return out

    def append(self, d):
        n = Node(d)
        if not self.head: n.next = n; self.head = n; return
        cur = self.head
        while cur.next is not self.head: cur = cur.next
        cur.next, n.next = n, self.head

    def prepend(self, d):
        n = Node(d, self.head)
        if not self.head: n.next = n; self.head = n; return
        tail = self.head
        while tail.next is not self.head: tail = tail.next
        tail.next = n; self.head = n

    def delete(self, key):
        h = self.head
        if not h: return False
        cur, prev = h, None
        while True:
            if cur.data == key:
                if prev: prev.next = cur.next
                else:
                    if cur.next is cur: self.head = None; return
True
                    tail = cur
                    while tail.next is not cur: tail = tail.next
                    self.head = cur.next; tail.next = self.head
                return True
            prev, cur = cur, cur.next
            if cur is h: return False

# demo
cll = CircularLinkedList()
cll.append(1); cll.append(2); cll.prepend(0)
print(cll.traverse())  # [0, 1, 2]
cll.delete(1)
print(cll.traverse())  # [0, 2]
```

Key Characteristics:

- Last node's `next`

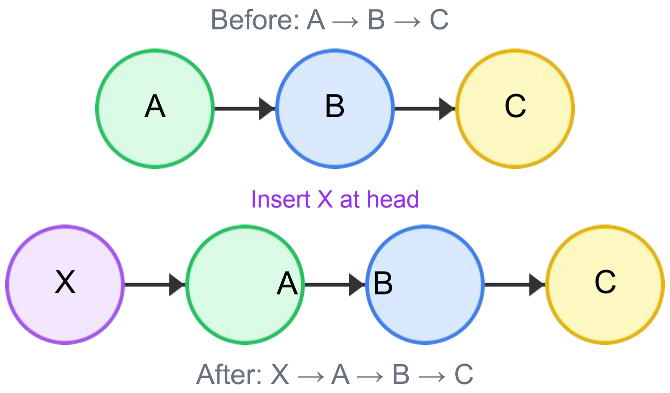
Insertion Operations

Inserting at Head

Inserting a new node at the beginning of the list. Updates the head pointer.

Time Complexity: O(1)

1. Create new node with data
2. Set new node's next to current head
3. Update head to new node

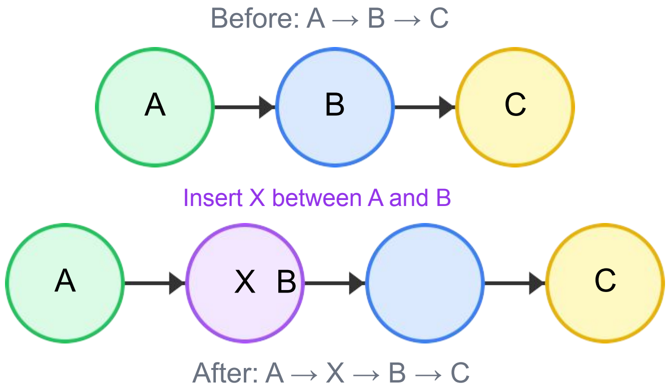


Inserting in Middle

Inserting a node between two existing nodes requires finding the insertion point.

Time Complexity: O(n)

1. Find node before insertion point
2. Create new node with data
3. Set new node's next to next node of previous
4. Set previous node's next to new node

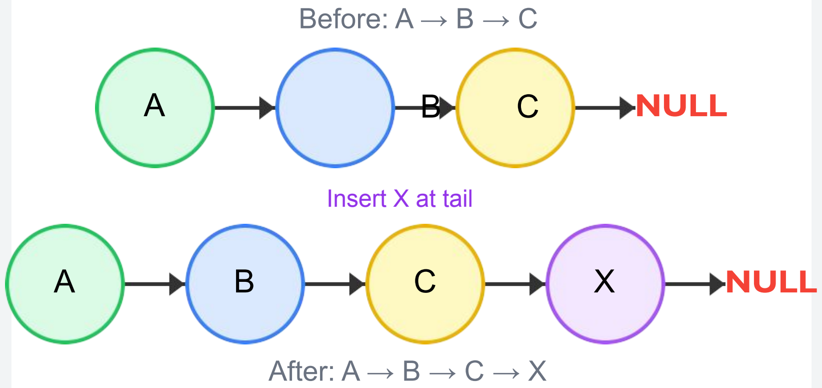


Inserting at Tail

Inserting a new node at the end of the list requires traversing to find the last node.

Time Complexity: O(n)

1. Traverse list to find last node
2. Create new node with data
3. Set last node's next to new node
4. Set new node's next to NULL



Note: Insertion at the head is the most efficient operation (O(1)), while middle and tail insertions require O(n) time due to traversal.

Deletion Operations

Deleting from Head

To delete the head node:

1. Update head

Point head to head.next.

2. Update tail's next

Find the last node (the one whose next points to old head).

Update its next to the new head.

3. Special case: only one node

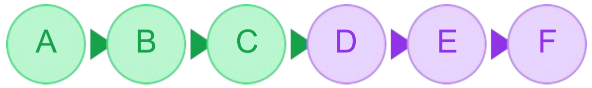
If head.next == head, then deleting the head makes the list empty.

```python
def delete_head(self):
    if self.is_empty():
        return

    # only one node
    if self.head.next == self.head:
        self.head = None
        return

    # find tail (node before head)
    tail = self.head
    while tail.next != self.head:
        tail = tail.next

    # move head forward
    self.head = self.head.next
    tail.next = self.head
```

Advantages of Linked Lists

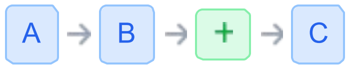
Dynamic Sizing

- Grows or shrinks during runtime without reallocation
- Memory allocated only as needed



Efficient Insertions/Deletions

- Insertion at beginning: O(1) time complexity
- Deletion at beginning: O(1) time complexity
- Deletion at end: O(1) for doubly linked lists



Memory Utilization

- More memory-efficient for dynamic data
- Avoids wasted space from oversized arrays
- Prevents expensive reallocations

Building Blocks

- Foundation for implementing stacks and queues
- Essential for hash tables and other structures

Stack Queue Hash Table

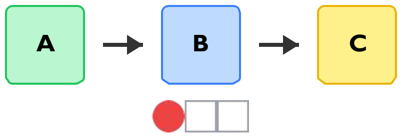
All built on linked list principles

Disadvantages of Linked Lists

Slow Random Access

Cannot access elements by index like arrays. Must traverse from the head node, resulting in:

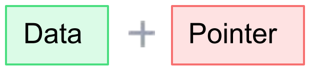
`O(n)` time complexity



Memory Overhead

Each node requires additional memory for pointer(s), increasing total memory usage compared to arrays.

For n elements: `n × (data_size + pointer_size)`



Poor Cache Performance

Non-contiguous memory storage leads to:

- Increased cache misses
- Reduced spatial locality

Key takeaway: Linked lists trade memory efficiency and cache performance for dynamic sizing and flexible insertion/deletion.

Real-World Applications

Linked lists are utilized in various everyday software applications where dynamic data management is required.

Music Player Playlists

Linked lists implement "next" and "previous" functionality, allowing seamless song navigation.

Previous Song | Current Song | Next Song

Browser History

Browsers use doubly linked lists for forward and backward navigation between visited pages.

Back | Home Page | Search | Forward

Undo/Redo Functionality

Applications use doubly linked lists for efficient undo and redo operations.

Undo | Edit | Cut | Redo

GPS Navigation Systems

Store and manage location lists and routes

Image Viewers

Implement "previous" and "next" image functionality

Computer Science Applications

Linked lists serve as foundational building blocks for implementing various abstract data types and algorithms.

Stack & Queue

Linked lists are ideal for implementing:

- **Stacks** (LIFO)
- **Queues** (FIFO)

Graph Representation

Linked lists enable efficient graph representation:

Adjacency List

Each vertex points to a list of adjacent vertices

Hash Table

Linked lists resolve hash collisions using:

Chaining

Multiple elements map to same index

Dynamic Memory

Linked lists help manage memory:

- OS uses linked lists to track free memory blocks
- Efficient allocation and deallocation

Polynomial

Linked lists represent polynomials:

Each node is a **term**

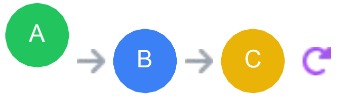
Stores coefficient and exponent

$3x^2 + 2x + 5$

Task Scheduling

Linked lists manage OS tasks:

- **Circular linked lists**
- Round-robin CPU time allocation



Implementing a Stack Using Linked Lists

Linked lists provide an excellent foundation for implementing stack data structures.

Key Implementation Points:

- Stack relies on **LIFO** (Last In, First Out) principle
- Linked list's **head** serves as the top of the stack
- **Push** operation inserts at the head (O(1))
- **Pop** operation removes from the head (O(1))

```
class Stack {

    private Node head;

    // Push operation
    public void push(int data) {
        Node newNode = new Node(data);
    newNode.next = head;
        head = newNode;
    }

    // Pop operation
    public int pop() {
        if (head == null) {
        throw new RuntimeException("Stack underflow");
        }
        int data = head.data;
        head     = head.next;
        return data;
    }

    // Peek operation
    public int peek() {
        if (head == null) {
        throw new RuntimeException("Stack underflow");
        }
        return head.data;
    }
}
```

Implementing a Queue Using Linked Lists

A queue is a FIFO (First In, First Out) data structure that can be efficiently implemented using linked lists.

Key Implementation Details:

- Maintain **head** and **tail** pointers
- Enqueue at the tail (O(1))
- Dequeue from the head (O(1))

python

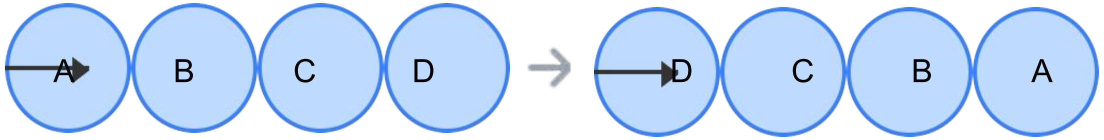
```python
class Node:
def  init (self, data):
    self.data = data
    self.next = None


class Queue:
    def  init (self):
    self.head = None
    self.tail = None

    def enqueue(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node self.tail =
    new_node
    else:
        self.tail.next = new_node self.tail =
    new_node


    def dequeue(self):
        if not self.head:
            return None
        data = self.head.data
        self.head = self.head.next
        if not self.head:
            self.tail = None
        return data
```

Advanced Linked List Operations

Beyond basic insertion and deletion, linked lists support several complex operations that demonstrate their versatility and power.

Reversal

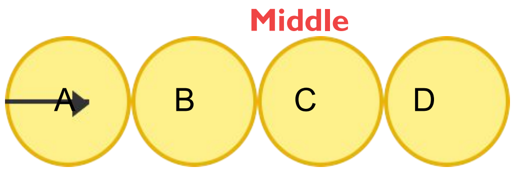
Reversing a linked list involves changing the direction of all pointers so that the last node becomes the head.



Key: Update pointers to previous node

Finding Middle Element

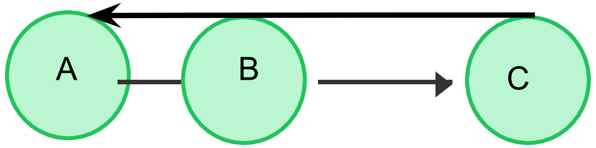
Finding the middle element can be efficiently done using the "tortoise and hare" approach with two pointers.



Key: One pointer moves twice as fast

Cycle Detection

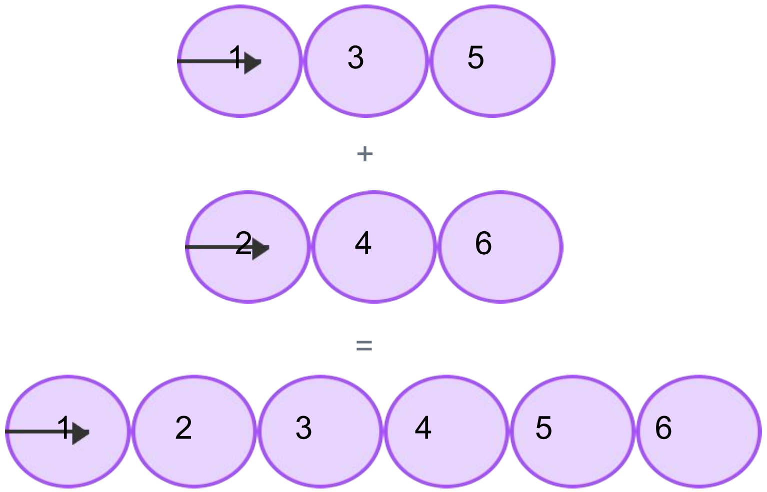
Detecting cycles is crucial for identifying circular references that can cause infinite loops.



Key: Use two pointers moving at different speeds

Merging Sorted Lists

Merging two sorted linked lists involves creating a new list in sorted order by comparing elements.



Key: Compare and link smaller elements

These operations demonstrate the versatility of linked lists beyond basic insertion and deletion.

Summary and Further Study

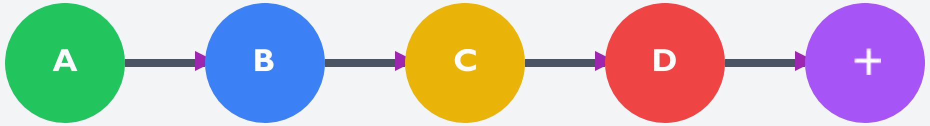
Key Concepts Recap

Structure
Nodes with data and pointers forming a sequence

Types
Singly, doubly, and circular linked lists

Operations
Insertion, deletion, traversal, searching

Applications
Browser history, playlists, undo/redo functionality



Further Study Suggestions

Advanced Data Structures
Trees and graphs built on linked list principles

Linked List Variations
Skip lists, XOR linked lists, and self-adjusting lists

Implementation Challenges
Real-world constraints and optimization techniques

Comprehensive Project Practice
Build complex systems, such as a music player or a browser history manager

> *"Mastering linked lists provides a crucial stepping stone for comprehending more advanced data structures."*

Report Date: 2025-09-03