Linked Lists

Nodes Connected via Pointers for Flexible Insertions/Deletions

Minseok Jeon
DGIST

November 2, 2025

Outline

Introduction

What is a Linked List?

Definition

A linear data structure where elements (nodes) are connected via pointers, allowing dynamic memory allocation and flexible insertions/deletions.

Key Characteristics:

- Non-contiguous memory storage
- Dynamic size (grows and shrinks at runtime)
- Each node contains data and pointer(s) to next/previous node(s)
- No random access (must traverse from head)

Why Learn Linked Lists?

- Foundation for stacks, queues, and other data structures
- Efficient insertions/deletions at known positions
- Understanding pointers and dynamic memory
- Common in interviews and real-world systems

Types of Linked Lists

Singly Linked List

Structure:

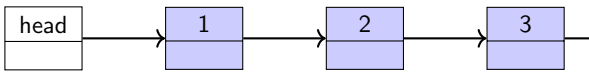
- Each node: data + next pointer
- Traversal: Forward only
- Memory: 1 pointer per node

Advantages:

- ✓Simple implementation
- ✓Less memory overhead
- ✓Efficient forward traversal

Disadvantages:

- × No backward traversal
- × Delete needs previous node
- × No direct tail access



Visual: head → [1|→] → [2|→] → [3|null]

Singly Linked List: Implementation

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

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return

        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def display(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        return elements
```

Doubly Linked List

Structure:

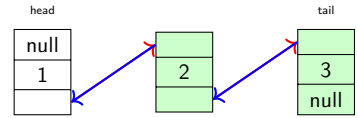
- Each node: data + next + prev pointers
- Traversal: Both directions
- Memory: 2 pointers per node

Advantages:

- ✓Bidirectional traversal
- ✓Easier deletion (no prev needed)
- ✓Can traverse from tail

Disadvantages:

- × More memory (2 pointers)
- × More complex implementation
- × Extra pointer maintenance



Visual:

head $\rightarrow$ [null|1|$\rightarrow$] $\leftrightarrow$ [$\leftarrow$|2|$\rightarrow$] $\leftrightarrow$ [$\leftarrow$|3|null] $\leftarrow$ tail

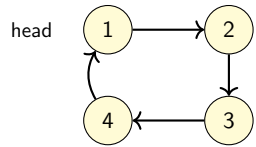
Doubly Linked List: Implementation

```python
class DNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def append(self, data):
        new_node = DNode(data)
        if not self.head:
            self.head = self.tail = new_node
            return

        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

    def delete(self, node):
        if node.prev:
            node.prev.next = node.next
        else:
            self.head = node.next

        if node.next:
            node.next.prev = node.prev
        else:
```

Circular Linked List

Structure:

- Last node points to first (cycle)
- Can be singly or doubly circular
- No natural end/null pointer



Advantages:

- ✓Traverse from any node
- ✓Round-robin scheduling
- ✓No null checks

Disadvantages:

- × Risk of infinite loops
- × Complex termination
- × Harder to detect end

Use Cases:

- Round-robin CPU scheduling
- Music playlists (repeat mode)
- Buffer management

Comparison Table

Feature	Singly	Doubly	Circular
Memory/node	1 pointer	2 pointers	1-2 pointers
Backward traversal	No	Yes	No (singly)
Delete with node	Need prev	$O(1)$	Need prev
Use case	Simple lists	Undo/redo	Round-robin

Head/Tail Pointers & Sentinels

Head and Tail Pointers

Head Pointer Only:

- Append: $O(n)$ (traverse to end)
- Prepend: $O(1)$
- Simpler, less memory

Head + Tail Pointers:

- Append: $O(1)$ (direct tail access)
- Prepend: $O(1)$
- Both ends accessible
- Perfect for queues

Benefits of Tail Pointer:

- ✓$O(1)$ append instead of $O(n)$
- ✓Efficient queue implementation
- ✓Direct access to last element

Trade-offs:

- Extra pointer to maintain
- Must update on append/delete
- Small memory overhead

Optimized List with Tail

```python
class OptimizedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def append(self, data):
        # O(1) with tail pointer
        new_node = Node(data)
        if not self.head:
            self.head = self.tail = new_node
            return

        self.tail.next = new_node
        self.tail = new_node

    def prepend(self, data):
        # O(1)
```

Sentinel (Dummy) Nodes

Concept

Use dummy nodes at head (and optionally tail) to eliminate edge cases for empty lists

Benefits:

- ✓Eliminates null checks for empty list
- ✓Simplifies insertion/deletion code
- ✓No special cases needed
- ✓Cleaner, more uniform code

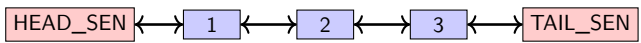
Trade-offs:

- × Extra memory for sentinel(s)
- × Slightly more complex initialization
- × Must skip sentinels during traversal

Sentinel Visualization

head $\rightarrow$ null

Without Sentinel: Empty list needs special handling



With Sentinels (Doubly Linked): Always valid, no edge cases

Sentinel Implementation

```python
class DoublyLinkedListWithSentinel:
    def __init__(self):
        self.head_sentinel = DNode(None)
        self.tail_sentinel = DNode(None)
        self.head_sentinel.next = self.tail_sentinel
        self.tail_sentinel.prev = self.head_sentinel

    def insert_before(self, node, data):
        # Always valid, no edge cases
        new_node = DNode(data)
        new_node.prev = node.prev
        new_node.next = node
        node.prev.next = new_node
        node.prev = new_node

    def delete(self, node):
        # Always valid, no edge cases
        node.prev.next = node.next
        node.next.prev = node.prev

    def is_empty(self):
        return self.head_sentinel.next == self.tail_sentinel
```

Insertion & Deletion Complexities

Insertion Operations

Insert at Beginning (Prepend):

- Create new node
- Point to current head
- Update head
- Time: $O(1)$

Insert at End (Append):

- Without tail: $O(n)$ (traverse)
- With tail: $O(1)$ (direct)

Insert at Position i:

- Traverse to position $i - 1$
- Create new node
- Update pointers
- Time: $O(i)$

Insert After Given Node:

- Have node reference
- Create new node
- Update pointers
- Time: $O(1)$

Insertion Examples

```python
def prepend(self, data):
    """Insert at beginning - O(1)"""
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node

def append_fast(self, data):
    """Insert at end with tail - O(1)"""
    new_node = Node(data)
    if not self.tail:
        self.head = self.tail = new_node
    else:
        self.tail.next = new_node
        self.tail = new_node

def insert_at(self, index, data):
    """Insert at position i - O(i)"""
    if index == 0:
        self.prepend(data)
        return

    current = self.head
    for _ in range(index - 1):
        if not current:
            raise IndexError("Index out of bounds")
        current = current.next

    new_node = Node(data)
    new_node.next = current.next
    current.next = new_node
```

Deletion Operations

Delete First Node:

- Move head to next
- Time: $O(1)$

Delete Last Node:

- Singly: $O(n)$ (find second-to-last)
- Doubly with tail: $O(1)$

Delete Node with Value:

- Search for node: $O(n)$
- Update pointers: $O(1)$
- Total: $O(n)$

Delete Given Node:

- Singly: Need previous node
- Doubly: $O(1)$ with node reference

Key Insight:

- Doubly linked lists excel at deletion
- Direct node access $\rightarrow$ $O(1)$ delete
- Singly linked requires traversal

Complexity Summary Table

Operation	Singly (no tail)	Singly (tail)	Doubly (tail)
Prepend	$O(1)$	$O(1)$	$O(1)$
Append	$O(n)$	$O(1)$	$O(1)$
Insert at i	$O(i)$	$O(i)$	$O(\min(i, n-i))$
Delete first	$O(1)$	$O(1)$	$O(1)$
Delete last	$O(n)$	$O(n)$	$O(1)$
Delete at i	$O(i)$	$O(i)$	$O(\min(i, n-i))$
Search	$O(n)$	$O(n)$	$O(n)$
Access by index	$O(i)$	$O(i)$	$O(\min(i, n-i))$

Reversal & Middle Finding

Reverse a Linked List: Iterative

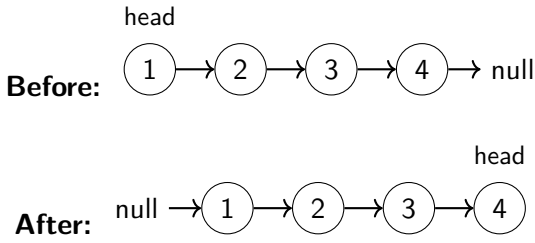
Algorithm:

1. Initialize prev = None, current = head
2. While current not None:
   - Save next_node = current.next
   - Reverse: current.next = prev
   - Move prev = current
   - Move current = next_node
3. Update head = prev

Complexity:

- Time: $O(n)$ - single pass
- Space: $O(1)$ - constant space

Reverse Visualization



Reverse Implementation

```python
def reverse_iterative(self):
    """Reverse linked list iteratively"""
    prev = None
    current = self.head

    while current:
        next_node = current.next  # Save next
        current.next = prev       # Reverse pointer
        prev = current            # Move prev forward
        current = next_node       # Move current forward

    self.head = prev
    # Time: O(n), Space: O(1)

def reverse_recursive(self, node):
    """Reverse linked list recursively"""
    if not node or not node.next:
```

Find Middle of Linked List

Two-Pointer (Slow-Fast) Technique:

- Use two pointers: slow and fast
- Slow moves one step at a time
- Fast moves two steps at a time
- When fast reaches end, slow is at middle

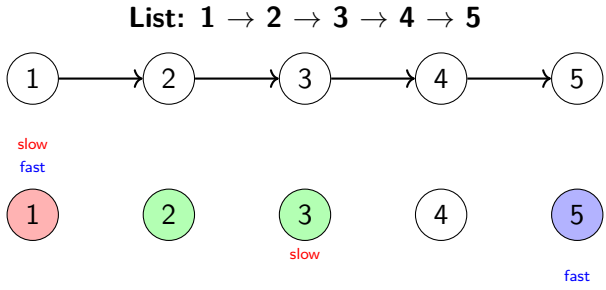
Complexity:

- Time: $O(n)$ - single pass
- Space: $O(1)$ - two pointers only

Advantages:

- No need to count nodes first
- Single traversal
- Works for odd and even length lists

Find Middle: Visualization



Middle: 3

Middle Finding & Cycle Detection

```python
def find_middle(self):
    """Find middle using slow-fast pointers"""
    if not self.head:
        return None

    slow = fast = self.head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    return slow  # Time: O(n), Space: O(1)

def has_cycle(self):
    """Detect cycle using Floyd's algorithm"""
    if not self.head:
        return False

    slow = fast = self.head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True  # Cycle detected

    return False

def find_kth_from_end(self, k):
    """Find k-th node from end"""
```

Comparison with Arrays

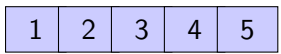
Linked Lists vs Arrays

Operation	Array	Linked List
Random access	$O(1)$	$O(n)$
Sequential access	$O(n)$	$O(n)$
Insert at beginning	$O(n)$	$O(1)$
Insert at end	$O(1)$ amortized	$O(1)$ with tail
Insert at position i	$O(n)$	$O(i)$
Delete at beginning	$O(n)$	$O(1)$
Delete at end	$O(1)$	$O(n)$ singly, $O(1)$ doubly
Search	$O(n)$ or $O(\log n)$	$O(n)$

Memory Layout Comparison

Array:

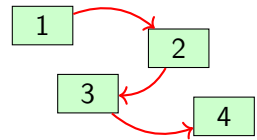
- Contiguous memory
- Address: base + i $\times$ sizeof(element)
- Better cache locality
- No pointer overhead



Contiguous block

Linked List:

- Scattered nodes
- Follow pointers
- Poor cache locality
- Pointer overhead



Scattered in memory

Advantages & Disadvantages

Arrays Advantages:

- ✓$O(1)$ random access
- ✓Better cache locality
- ✓Less memory overhead
- ✓Binary search possible
- ✓Better for iteration

When to Use Arrays:

- Frequent random access
- Data size known
- Memory locality important
- Need sorting/searching
- Iteration primary operation

Linked Lists Advantages:

- ✓$O(1)$ insert/delete at known position
- ✓No wasted space
- ✓No element shifting
- ✓Easy split/merge
- ✓No reallocation

When to Use Linked Lists:

- Frequent insertions/deletions
- Unknown/variable size
- No random access needed
- Implementing stacks/queues
- Need to split/merge

Memory Overhead & Locality

Memory Overhead Analysis

Example: 5 integers

- **Array:**
  - Memory: 5×4 bytes $= 20$ bytes (data only)
  - Small overhead for array metadata
- **Singly Linked List:**
  - Per node: 4 bytes (int) + 8 bytes (pointer) $= 12$ bytes
  - Total: $5 \times 12 = 60$ bytes
  - Overhead: **200% more** than array!
- **Doubly Linked List:**
  - Per node: 4 bytes (int) + 16 bytes (2 pointers) $= 20$ bytes
  - Total: $5 \times 20 = 100$ bytes
  - Overhead: **400% more** than array!

Key Insight

Linked lists have significant memory overhead due to pointers

Cache Locality Impact

Array (Good Locality):

- Contiguous layout: [1][2][3][4][5]
- Cache line (64 bytes): loads multiple elements
- Sequential access → cache-friendly
- Fast iteration

Performance:

- Most accesses hit cache (L1/L2)
- Prefetching effective
- Typical: 1-4 ns per element

Linked List (Poor Locality):

- Scattered: [1|→] ... [2|→] ... [3|→] ...
- Cache line: only one node
- Pointer chasing → cache misses
- Slow iteration

Performance:

- Frequent cache misses
- Prefetching ineffective
- Typical: 5-10x slower than array

Real-World Impact

For iteration over 1 million elements, linked lists can be 5-10x slower due to cache misses

Optimization Techniques

Improving Linked List Performance:

1. **Memory Pools:**
   - Allocate nodes from contiguous pool
   - Better cache locality
   - Reduces fragmentation
2. **Unrolled Linked Lists:**
   - Store multiple elements per node
   - Array-like access within node
   - Balance between linked list and array
3. **Skip Lists:**
   - Add express lanes for faster search
   - $O(\log n)$ search instead of $O(n)$
   - Probabilistic data structure
4. **XOR Linked Lists:**
   - Store XOR of prev and next addresses

Real-World Applications

Application 1: Stack & Queue Implementation

Stack (LIFO):

- Push: $O(1)$ at head
- Pop: $O(1)$ from head
- Simple, efficient

Operations:

- push(x): Add to head
- pop(): Remove from head
- peek(): View head

Queue (FIFO):

- Enqueue: $O(1)$ at tail
- Dequeue: $O(1)$ from head
- Need head and tail pointers

Operations:

- enqueue(x): Add to tail
- dequeue(): Remove from head
- front(): View head

Why Linked Lists?

Both ends accessible in $O(1)$, dynamic size, no reallocation

Application 2: Browser History

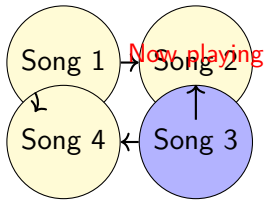
Back/Forward Navigation:

- Use doubly linked list
- Current page = current node
- Back button: current = current.prev
- Forward button: current = current.next
- Visit new page: add node after current, clear forward history

Application 3: Music Playlist

Circular Linked List for Playlists:

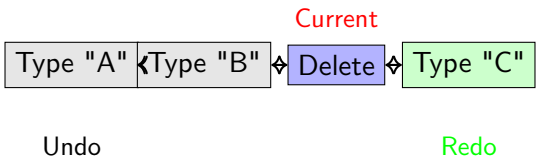
- Last song points to first (repeat mode)
- Current song = current node
- Next song: current = current.next
- Previous song: traverse or use doubly circular
- Add/remove songs dynamically



Application 4: Undo/Redo Functionality

Text Editor Undo/Redo:

- Use doubly linked list of actions
- Current = current action
- Undo: current = current.prev, revert action
- Redo: current = current.next, apply action
- New action: add after current, clear redo history



Application 5: LRU Cache

Least Recently Used Cache:

- Doubly linked list + hash table
- List ordered by recency (head = most recent, tail = least recent)
- Hash table: key $\rightarrow$ node (for $O(1)$ access)
- Get: move accessed node to head
- Put: add to head, if full, remove tail

Complexity:

- Get: $O(1)$ (hash lookup + move to head)
- Put: $O(1)$ (hash insert + add to head)

Use Cases:

- Browser caching
- Database query cache
- Operating system page replacement

Other Real-World Applications

1. **Operating Systems:**
   - Process scheduling queues (ready, waiting)
   - Memory management (free lists)
   - File system directory entries
   - Device driver I/O queues
2. **Hash Table Chaining:**
   - Each bucket is a linked list
   - Handle collisions efficiently
   - Dynamic size per bucket
3. **Symbol Tables:**
   - Compiler symbol tables
   - Variable scope management
   - Function call stack frames
4. **Graph Representations:**
   - Adjacency lists for graphs
   - Each vertex has linked list of neighbors

Summary

Key Takeaways

Types of Linked Lists:

- Singly: Simple, one-way traversal
- Doubly: Bidirectional, easier deletion
- Circular: No end, round-robin scheduling

Optimization Techniques:

- Tail pointer: $O(1)$ append
- Sentinel nodes: Eliminate edge cases
- Two-pointer: Efficient middle finding, cycle detection

Trade-offs:

- $O(1)$ insertion/deletion at known positions vs $O(n)$ random access
- Dynamic size vs memory overhead (pointers)
- Flexibility vs poor cache locality

When to Use Linked Lists

Choose Linked Lists When:

- Frequent insertions/deletions at arbitrary positions
- Size unknown or highly variable
- No need for random access
- Implementing stacks, queues, or other ADTs
- Need to frequently split or merge lists

Choose Arrays When:

- Need frequent random access
- Size is known or stable
- Memory locality is critical
- Need to sort or perform binary search
- Primarily iterating over elements

Key Principle

Choose the data structure that optimizes for your most frequent operations

Practice Problems

Essential Linked List Problems:

1. Reverse a linked list (iterative and recursive)
2. Detect cycle in linked list (Floyd's algorithm)
3. Find middle of linked list (two pointers)
4. Merge two sorted linked lists
5. Remove n-th node from end
6. Check if linked list is palindrome
7. Find intersection of two linked lists
8. Remove duplicates from sorted/unsorted list
9. Add two numbers represented as linked lists
10. Flatten a multilevel doubly linked list

Resources:

- LeetCode: Linked List problems (Easy to Hard)
- GeeksforGeeks: Comprehensive linked list articles
- Cracking the Coding Interview: Chapter on Linked Lists

Thank You!

Questions?

"A linked list is like a treasure hunt – each node points you to the next clue!"

Master linked lists and you'll understand the power of pointers!