Circular and Header Linked Lists

Data Structures & Algorithms

Class 03

Topic: Circular Lists & Header Node Variations

Learning Objectives

By the end of this class, you will be able to:

1. Understand the structure and purpose of circular linked lists
2. Differentiate between circular singly and circular doubly linked lists
3. Implement operations on circular linked lists
4. Understand header nodes and their advantages
5. Recognize real-world applications of circular structures
6. Choose the appropriate linked list type for specific problems

Recap: The Journey So Far

Class 01: Introduction to Linked Lists

- Basic concept of nodes and pointers
- Simple linked list creation
- Why linked lists exist

Class 02: ADT and List Types

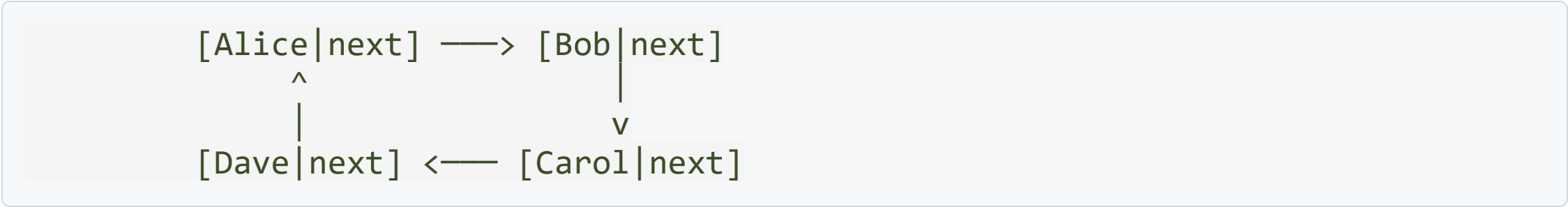
- Abstract Data Types (ADT) - the "what" vs "how"
- Singly Linked Lists - one-way navigation
- Doubly Linked Lists - bidirectional travel
- Core operations (insert, delete, traverse)

Today: Completing the linked list family tree!

Circular Linked Lists: Where the Circle Never Ends

A Story to Begin

Imagine a system where a **token** is passed between four nodes: **Alice**, **Bob**, **Carol**, and **Dave**.



In a standard linear list, the sequence terminates when a node's `next` pointer is `NULL`. In this model, Dave's `next` pointer stores the memory address of Alice. This creates a continuous loop where traversal never encounters a termination signal.

This is exactly how a **Circular Linked List** works!

What is a Circular Linked List?

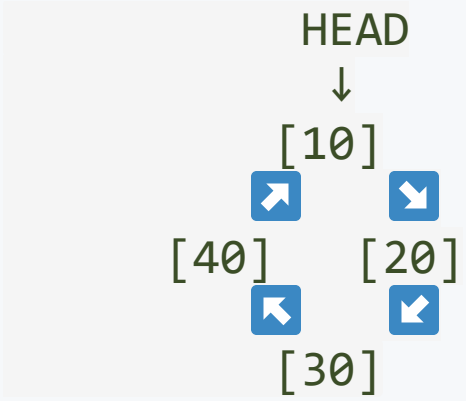
Circular Linked List: A variation where:

- The **last node** doesn't point to `NULL`
- Instead, it **points back to the first node**
- Creates a **circular chain** with no natural endpoint

Key Difference:

```
Singly Linked List:
HEAD → [10|•] → [20|•] → [30|•] → [40|NULL]
                                  ↑ Ends here!

Circular Linked List:
       ┌──────────────────────────────────┐
       ↓                                  │
HEAD → [10|•] → [20|•] → [30|•] → [40|•]──┘
       ↑ Can start anywhere, never ends!
```

Visual Circle Representation



Every node is both a beginning and an end!

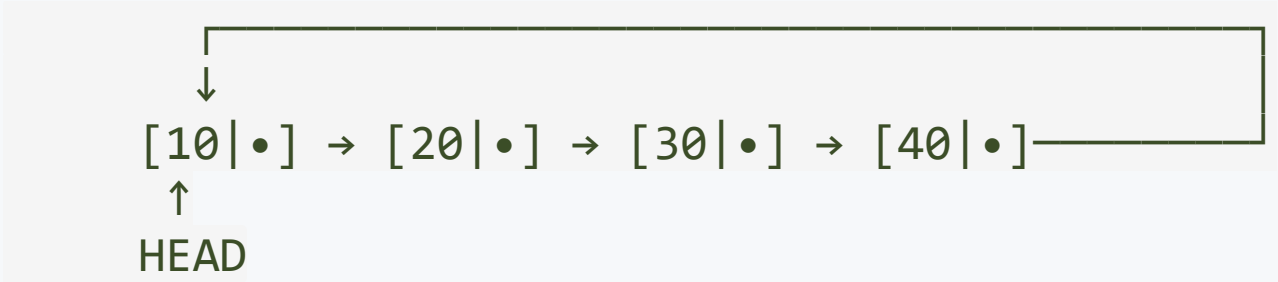
Types of Circular Linked Lists

Type 1: Circular Singly Linked List

Each node has:

- **Data** field
- **One pointer** to the next node
- **Last node** points back to the first node

Structure:



Key Point: Can only move forward, but never reaches NULL!

Memory View: Circular Singly

```
HEAD points to: Address 1000

Node at 1000: [10 | 2500]  →  Points to 2500
Node at 2500: [20 | 7800]  →  Points to 7800
Node at 7800: [30 | 1200]  →  Points to 1200
Node at 1200: [40 | 1000]  →  Points back to 1000!
                      ↑
             Completes the circle
```

How to detect the end?

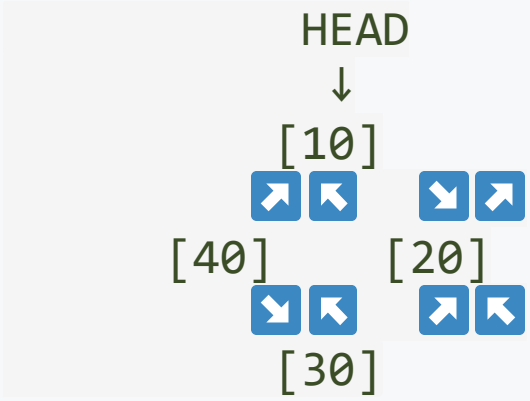
```
// Regular list: check for NULL
while (current != NULL)  // Won't work! Never NULL!

// Circular list: check if back to start
while (current->next != HEAD)  // Back to beginning!
```

Type 2: Circular Doubly Linked List

Each node has:

- **Data** field
- **Two pointers:** `next` (forward) and `prev` (backward)
- **Last node's next** → first node
- **First node's prev** → last node

Simplified Circle View:



Each arrow is bidirectional (prev/next)

Memory View: Circular Doubly

```
Node at 1000: [1200 | 10 | 2500]  ← First node
               ↑ prev   data  next ↓
            Points to          Points to
            last (1200)        second (2500)

Node at 2500: [1000 | 20 | 7800]
Node at 7800: [2500 | 30 | 1200]
Node at 1200: [7800 | 40 | 1000]  ← Last node
                          next ↑
                   Points back to first!
```

Navigation Power:

- Go forward: `current = current->next;` (Clockwise)
- Go backward: `current = current->prev;` (Counter-clockwise)
- Never hits NULL in either direction!

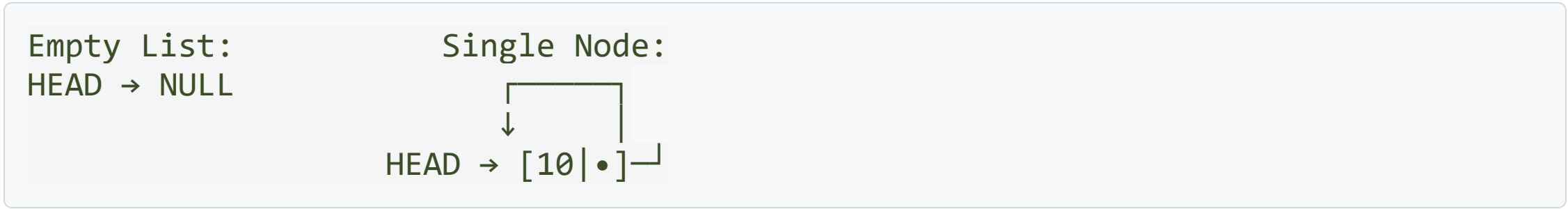
Operations on Circular Linked Lists

Checking if List is Empty

Challenge: No NULL to check! How do we know if empty?

```
// Method 1: Check if HEAD is NULL
if (HEAD == NULL) {
    // List is empty
}

// Method 2: If HEAD exists, check if it points to itself
if (HEAD != NULL && HEAD->next == HEAD) {
    // Single node circular list
}
```

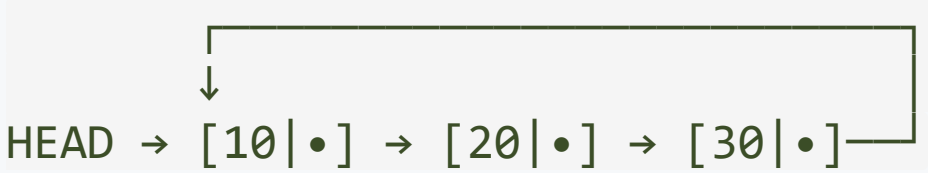
Visual:



Insert at Beginning - Overview

Goal: Insert `5` at the start

Starting point:



Challenge: The last node points to the current first node. We need to:

1. Make new node point to current first
2. Make last node point to new node
3. Update HEAD

Time Complexity: O(n) - Must traverse to find last node

Optimization: Maintain a `TAIL` pointer → becomes O(1)!

Insert at Beginning - Steps

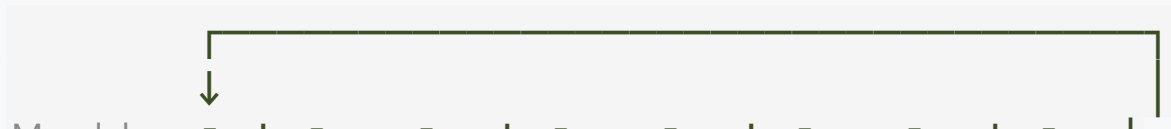
Step 1: Find the last node

```
struct Node* last = HEAD;
while (last->next != HEAD) {
    last = last->next;
}
// Now 'last' points to the last node
```

Step 2: Make connections

```
newNode->next = HEAD;      // New points to old first
last->next = newNode;      // Last points to new first
HEAD = newNode;            // Update HEAD
```

Result:



Insert at End (Circular Singly)

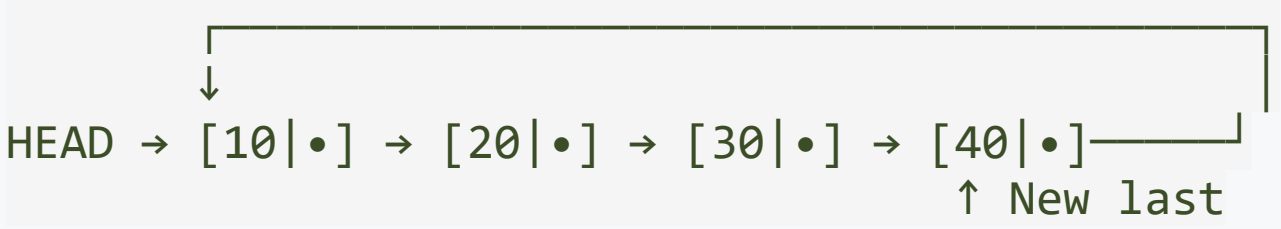
Goal: Insert `40` at the end

Step 1: Find the last node (same as before)

Step 2: Create new node and connect

```
newNode = createNode(40);
newNode->next = HEAD;       // New node points to first
last->next = newNode;       // Old last points to new node
// HEAD remains unchanged
```

Result:

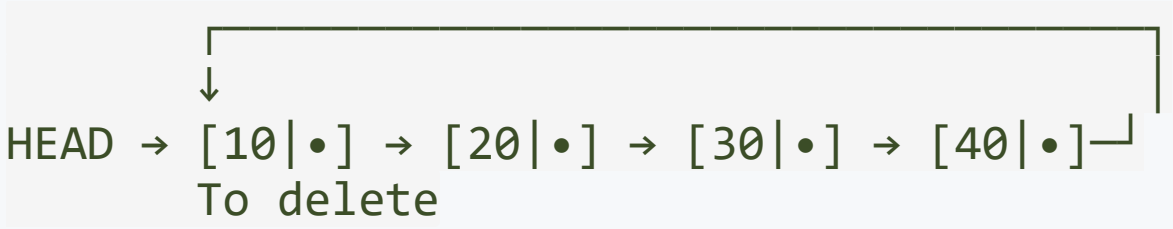


Time Complexity: O(n) - Must find last node

Delete from Beginning

Goal: Remove first node

Starting:



Steps:

```
struct Node* last = HEAD;
while (last->next != HEAD) {
    last = last->next;
}

struct Node* temp = HEAD;
last->next = HEAD->next;   // Last now points to second
HEAD = HEAD->next;         // HEAD moves to second
free(temp);                // Free old first
```

Traversal (Circular Linked List)

Challenge: When to stop? There's no NULL!

Strategy: Stop when you return to HEAD

```
void printCircular(struct Node* HEAD) {
    if (HEAD == NULL) {
        printf("Empty list\n");
        return;
    }

    struct Node* current = HEAD;

    // Use do-while to print at least first node
    do {
        printf("%d → ", current->data);
        current = current->next;
    } while (current != HEAD);

    printf("(back to %d)\n", HEAD->data);
}
```

Traversal - Visual Execution

```
Step 1: current → [10]  Print: 10
Step 2: current → [20]  Print: 20
Step 3: current → [30]  Print: 30
Step 4: current → [40]  Print: 40
Step 5: current → [10]  Back to HEAD! STOP!

Output: 10 → 20 → 30 → 40 → (back to 10)
```

Common Pitfall:

```c
// WRONG - infinite loop!
while (current != NULL) {  // Never NULL!
    printf("%d ", current->data);
    current = current->next;
}

// CORRECT
while (current->next != HEAD) {  // Check return to HEAD
    printf("%d ", current->data);
    current = current->next;
}
```

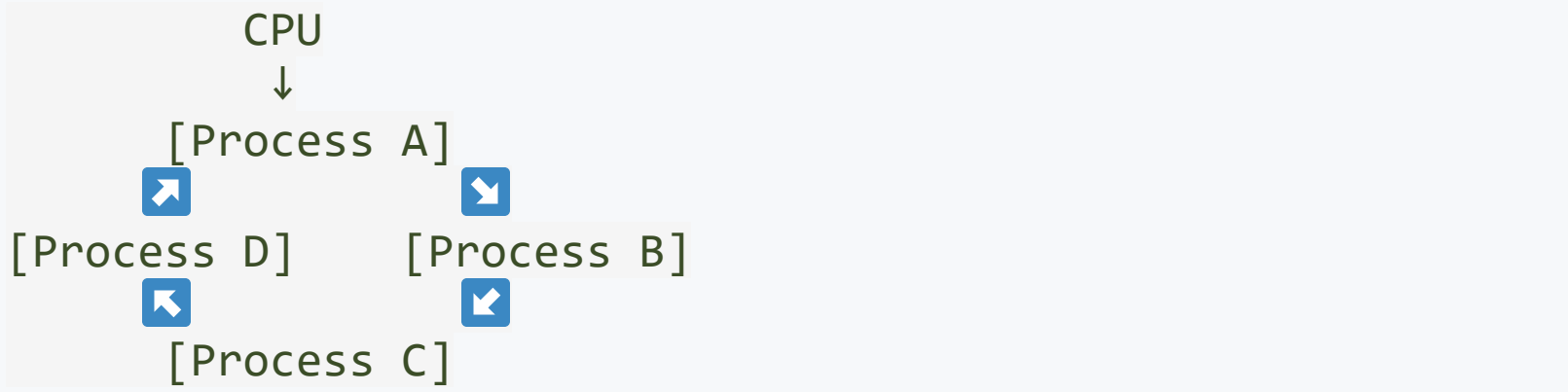
Advantages of Circular Linked Lists

Advantage	Explanation	Use Case
No NULL Checks	Never encounter NULL pointers	Simpler algorithms
Reach Any Node from Any	Can start anywhere, reach everywhere	Round-robin scheduling
Efficient for Cyclic Ops	Natural for repeating processes	Media players, games
No Special Case for Last	Last node is just another node	Cleaner code

Real-World Applications

Application 1: CPU Task Scheduling

Operating systems use circular lists for fair CPU time allocation:




```
Each process gets CPU for a time slice, then moves to next.
After D, back to A. Fair and infinite!
```

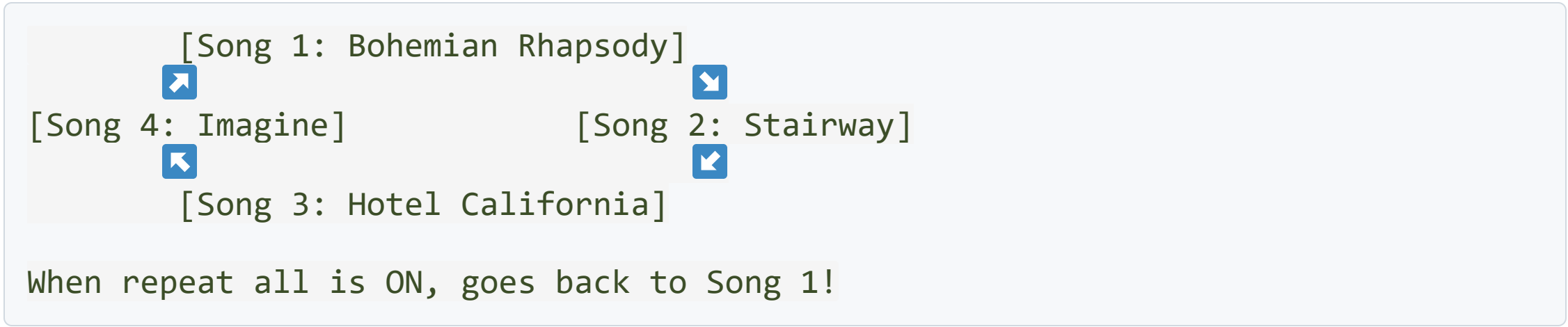
Round-Robin Scheduling: Equal time for all processes

Application 2: Multiplayer Board Games

```
    [Player 1: Alice] → [Player 2: Bob]
           ↑                      ↓
    [Player 4: Dave] ← [Player 3: Carol]

After Dave's turn → Back to Alice
Perfect for "whose turn is it?" logic!
```

Natural representation for turn-based systems

Application 3: Music Playlist (Repeat Mode)



Continuous playback with repeat functionality

Application 4: Circular Buffer

Used in streaming, audio processing, networking:

Applications:

- Audio/video streaming
- Keyboard buffers
- Network packet buffers

Application 5: Josephus Problem

Famous algorithm problem:

```
N people stand in a circle. Starting from a point,
count K people and eliminate the Kth person.
Continue until one person remains.

Example: N=5, K=2
Start: 1 → 2 → 3 → 4 → 5 → (back to 1)
Count 2: Eliminate 2
Remaining: 1 → 3 → 4 → 5 → (back to 1)
Count 2 from 3: Eliminate 4
... continues until one survives!
```

Circular linked list makes this elegant to implement!

Header Nodes: The Special Sentinel

What is a Header Node?

A **Header Node** (or **Dummy Node**) is a special node at the beginning that:

- **Doesn't store actual data** (or stores metadata)
- **Always exists**, even in an "empty" list
- **Simplifies operations** by eliminating special cases

Analogy: Like a team captain who doesn't play but coordinates the players!

Structure Comparison

Regular Linked List:

```
Head pointer → [10|•] → [20|•] → [30|NULL]
                ↑
          First real data node
```

With Header Node:

```
Head pointer → [DUMMY|•] → [10|•] → [20|•] → [30|NULL]
                  ↑           ↑
              Header Node  First real data
          (always exists)
```

Key Difference:

- Regular: Head points to first data node (or NULL if empty)
- Header: Head always points to header node (never NULL!)

Types of Header Nodes

Type 1: Dummy Header

```c
struct Node {
    int data;           // Ignored in header
    struct Node* next;
};

struct Node* createListWithHeader() {
    struct Node* header = malloc(sizeof(struct Node));
    header->data = -1;   // Dummy value (not used)
    header->next = NULL; // Empty list
    return header;
}
```

Visual:

```
Empty list:
HEAD → [DUMMY|NULL]

List with data:
HEAD → [DUMMY|•] → [10|•] → [20|NULL]
```

Type 2: Grounded Header List

Header contains **metadata** about the list:

```
struct HeaderNode {
    int count;          // Number of nodes in list
    struct Node* next; // Points to first data node
};
```

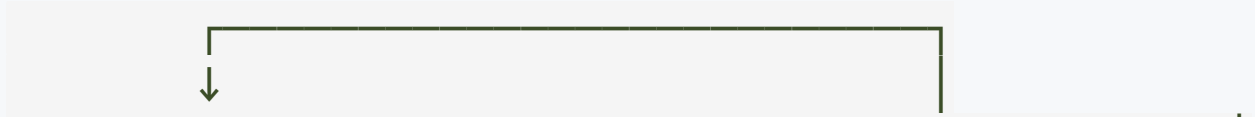
Visual:

```
HEAD (HeaderNode)
  ├── count: 3
  └── next → [10|•] → [20|•] → [30|NULL]

Instantly know size without traversal!
```

Benefit: O(1) access to list statistics!

Type 3: Circular Header List

Combines header node with circular structure:



Last node points back to header, not first data node!

Advantages of Header Nodes

Advantage 1: Eliminates Empty List Special Case

Without Header:

```
void insert(struct Node** head, int val) {
    struct Node* newNode = createNode(val);

    if (*head == NULL) {            // Special case!
        *head = newNode;
    } else {
        newNode->next = *head;
        *head = newNode;
    }
}
```

With Header:

```
void insert(struct Node* header, int val) {
    struct Node* newNode = createNode(val);
    newNode->next = header->next;  // Always works!
    header->next = newNode;        // Even if empty!
}
// No if-else needed!
```

Advantage 2: Simplified Deletion

Without Header - special case for first node:

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

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
```

With Header - uniform operation:

```
void deleteFirst(struct Node* header) {
    if (header->next == NULL) return;

    struct Node* temp = header->next;
    header->next = temp->next;
    free(temp);
}
```

Advantage 3: Cleaner Position-Based Operations

Without header: Position 0 requires special handling

With header: Position 0 is just another position!

```
void insertAtPosition(struct Node* header, int val, int pos) {
    struct Node* current = header;  // Start from header

    // Traverse to position
    for (int i = 0; i < pos; i++) {
        if (current == NULL) return;
        current = current->next;
    }

    // Insert after 'current' - works for ALL positions!
    struct Node* newNode = createNode(val);
    newNode->next = current->next;
    current->next = newNode;
}
```

Advantage 4: Metadata Storage

```
struct HeaderNode {
    int count;      // Size of list
    int maxVal;     // Track maximum
    int minVal;     // Track minimum
    struct Node* next;
};

// Get size in O(1) instead of O(n)!
int getSize(struct HeaderNode* header) {
    return header->count;  // Instant!
}

// Update count during insert
void insert(struct HeaderNode* header, int val) {
    // ... insertion logic ...
    header->count++;        // Update metadata
    if (val > header->maxVal) header->maxVal = val;
}
```

Benefit: O(1) access to list statistics!

Disadvantages of Header Nodes

Disadvantage	Impact
Extra Memory	One additional node always allocated
Slightly More Complex	Need to remember header exists
Wasted Space	Header's data field unused (in dummy variant)

Trade-off: Extra memory for simpler, cleaner code!

Comparison: All Linked List Types

Comparison Table

Feature	Singly	Doubly	Circular Singly	Circular Doubly	With Header
Forward Traversal	Easy	Easy	Easy	Easy	Easy
Backward Traversal	No	Easy	Must go around	Easy	Depends
Insert at Beginning	O(1)	O(1)	O(n)*	O(1)	O(1)
Insert at End	O(n)	O(1)**	O(n)	O(1)**	O(n)
Delete at Beginning	O(1)	O(1)	O(n)*	O(1)	O(1)
Memory per Node	1 ptr	2 ptrs	1 ptr	2 ptrs	+1 node

Choosing the Right List Type

Decision Guide

Use Singly Linked List when:

- Memory is tight
- Only forward traversal needed
- Implementing stack (LIFO)
- Simple applications

Use Doubly Linked List when:

- Need backward traversal
- Frequent deletions (have node reference)
- Implementing deque (double-ended queue)
- Browser history (back/forward)

Decision Guide (continued)

Use Circular Linked List when:

- Round-robin scheduling
- Cyclic processes (games, playlists)
- No beginning/end concept
- Implementing circular buffer

Use Header Node when:

- Want cleaner, uniform code
- Need to store metadata (size, max, min)
- Frequent empty list operations
- Avoiding special case handling

Hybrid Solutions

Circular + Doubly: Media players with repeat + rewind

Circular + Header: OS process scheduling with stats

Doubly + Header: Complex data structures, databases

Choose based on:

- Memory constraints
- Operation frequency
- Code simplicity needs
- Performance requirements

Advanced Topic: Floyd's Cycle Detection

Detecting Loops in Regular Lists

Famous Interview Question: How to detect if a linked list has a cycle?

Floyd's Cycle Detection (Tortoise and Hare):

```
int hasCycle(struct Node* head) {
    if (head == NULL) return 0;

    struct Node* slow = head;
    struct Node* fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;          // Move 1 step
        fast = fast->next->next;  // Move 2 steps

        if (slow == fast) {
            return 1;  // Cycle detected!
        }
    }

    return 0;  // No cycle (reached NULL)
}
```

Why Does This Work?

```
If there's a cycle, fast will eventually catch up to slow!
Like a race track - faster runner will lap slower runner.

No cycle:          With cycle:
1→2→3→NULL        1→2→3
                      ↑_↓
                        4
slow: 1,2,3,NULL
fast: 2,NULL      slow: 1,2,3,4,2,3,4...
No meeting!       fast: 2,4,3,2,4...
                       ↑_____↑ They meet!
```

Time Complexity: O(n)

Space Complexity: O(1) - Only two pointers!

Summary

Key Takeaways

Circular Linked Lists:

- Last node points back to first (or header)
- Two types: Circular Singly and Circular Doubly
- No NULL pointers - check return to HEAD instead
- Perfect for cyclic operations (scheduling, playlists, games)
- Insertion/deletion more complex (must maintain circle)

Key Takeaways (continued)

Header Nodes:

- Special sentinel node that always exists
- Three types: Dummy, Grounded (with metadata), Circular
- Eliminates empty list special cases
- Enables O(1) size/metadata access
- Cleaner, more uniform code

Main Lesson:

- Different problems need different list structures
- Circular lists excel at repeating processes
- Header nodes trade memory for code simplicity
- Can combine features (circular + doubly + header)
- Understanding trade-offs helps you choose wisely

Next Class Preview

Stack and Queues: Two ADTs implemented using linked lists!

- Stack (LIFO): Last In, First Out
- Queue (FIFO): First In, First Out
- Practical applications
- Implementation using various linked list types

Keep Learning! Keep Coding! Keep Linking!