8

Circular Linked Lists

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

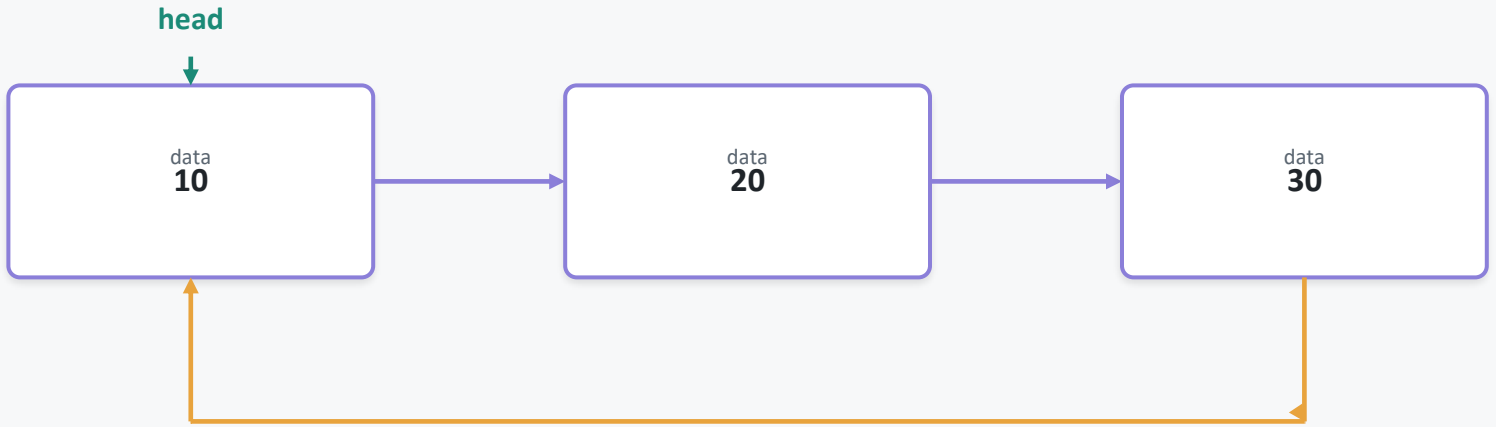
Agenda / Learning Objectives

1. **The circular concept**
   How head and tail change meaning when there's no nullptr
2. **Linear vs. circular, side by side**
   One arrow is the entire structural difference
3. **Traversal, insertion, deletion**
   Why do-while replaces while, with pointer-rewiring diagrams
4. **Pitfalls, and round-robin scheduling**
   A fully worked CPU scheduler built on the ring
5. **Singly vs. doubly vs. circular**
   A direct three-way comparison to close the unit

CORE IDEA

Head and Tail in a Circular Linked List

There is no nullptr anywhere in a non-empty circular list — the last node's next points back to head. Every traversal must check "have I gotten back to where I started?"

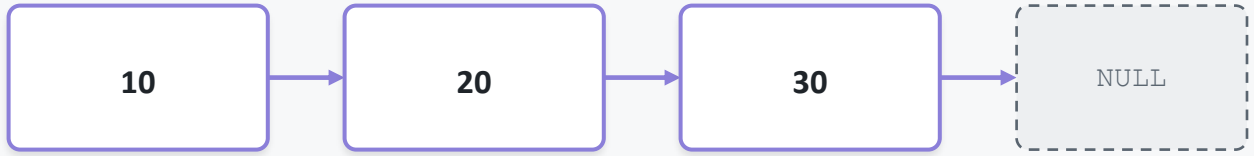


CORE IDEA

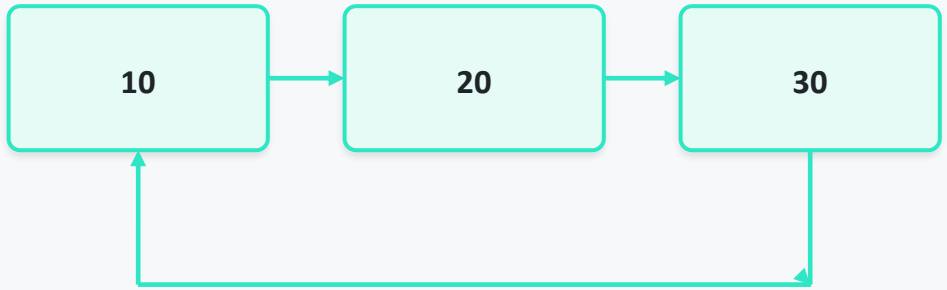
Side by Side: Linear vs. Circular

The node structure is identical — the difference is a single arrow, and that arrow removes the concept of "the end" entirely.

Singly linked list — has an end



Circular linked list — no end



wraps around

Any loop written as while (current != nullptr) must become do { ... } while (current != startingPoint) — there is no sentinel to stop on.

DESIGN

The CircularLinkedList Class

Tracking tail (not head) makes end-insertion simpler here — tail->next is always the head.

circular_linked_list.cpp

```
void insertAtBeginning(int value) {
    Node* newNode = new Node(value);
    if (isEmpty()) {
        tail = newNode;
        tail->next = tail;   // self-loop!
        return;
    }
    newNode->next = tail->next;
    tail->next = newNode;
}
```

tail, not head

The class's one stored pointer — tail->next always gives the head.

insertAtEnd reuses this

insertAtBeginning(value); then tail = tail->next — slide tail forward by one.

traverse() uses do-while

Runs the body once before checking current != head, or nothing would print.

EDGE CASE

The One-Node Self-Loop

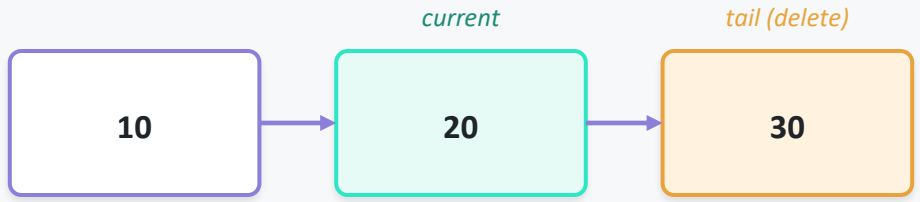
The very first insertion into an empty circular list is its own edge case: a single node's next must point at itself, not nullptr.

Traversal & Deletion From the End

traverse() output

```
$ ./circular_linked_list
10 -> 20 -> 30 -> (back to 10)
```

deleteFromEnd walks the whole ring — no prev pointer, no shortcut to the second-to-last node.




```
current->next = tail->next;
// skip over old tail, back to head
delete tail;
tail = current;
```

O(n)

deleteFromEnd — tail helps with insertion, not with finding the second-to-last node

Common Pitfalls With Circular Lists

while (current != nullptr) out of habit

There is no nullptr to stop on — this loop simply never terminates.

Checking the stop condition too early

A plain while (current != head) starts false and skips everything — do-while is required.

Forgetting the one-node self-loop

insertAtBeginning on an empty list must set tail->next = tail.

Losing tail after deleting it

Deleting the only node must reset tail to nullptr, or the list claims to be non-empty.

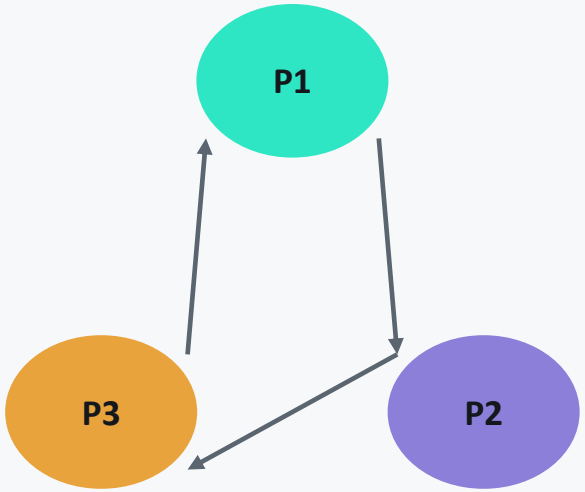
WORKED EXAMPLE

Application: Round-Robin CPU Scheduling

Every process gets a fixed time slice (a quantum); the ring structure IS the wraparound — no special "restart" logic needed.

```
quantum = 4, P1=5 P2=3 P3=7

$ ./round_robin_scheduler
t=0: run P1 for 4 (remaining: 1)
t=4: run P2 for 3 (remaining: 0)
        P2 finished.
t=7: run P3 for 4 (remaining: 3)
t=11: run P1 for 1 (remaining: 0)
        P1 finished.
t=12: run P3 for 3 -- P3 finished.
All processes finished at t=15.
```



ring order

WHY IT MATTERS

Applications of Circular Linked Lists

Round-robin CPU scheduling

Each process gets a turn; after the last, the scheduler wraps back to the first, forever.

Multiplayer turn order

After the last player's turn, play returns to the first player.

Looping playlists

"Repeat all" needs the last song to lead straight back into the first.

Streaming data buffers

A fixed-size circular buffer reuses the same nodes instead of endlessly allocating.

COMPARISON

Singly vs. Doubly vs. Circular

	Singly	Doubly	Circular (singly)
Pointers per node	1 (next)	2 (prev, next)	1 (next)
Last node's next	nullptr	nullptr	Points back to head
Backward traversal	No	Yes	No
Natural "end"	Yes	Yes	No -- track a start point
Typical use case	General-purpose	Need both directions	Looping / round-robin

KT Key Takeaways

- A circular linked list's last node points back to head instead of nullptr — there is no natural end; every traversal must detect "back to the start."
- Tracking tail (rather than head) makes end insertion O(1), since tail->next is always the head.
- The very first node inserted into an empty circular list must point at itself — a self-loop is the only correct structure for one node.
- do-while is the standard loop shape for circular traversal — a while loop would wrongly treat "already at the start" as "already done."
- Circular linked lists are the right tool specifically when the problem itself loops — round-robin scheduling, turn-based games, repeating playlists.