7

Doubly Linked Lists

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

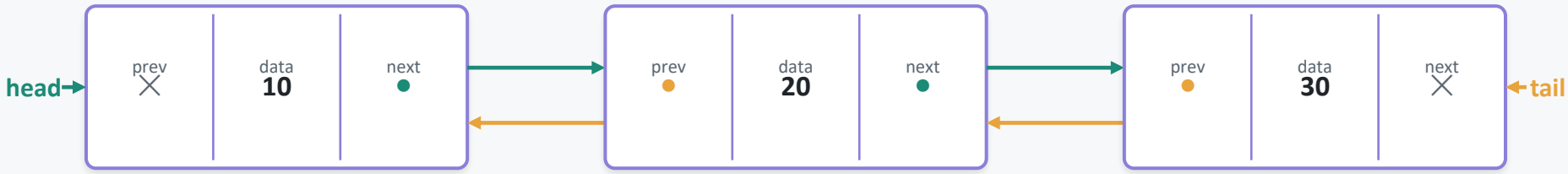
Agenda / Learning Objectives

1. **The doubly linked node**
   A three-field struct: prev, data, and next
2. **Forward & backward traversal**
   Including a step-by-step worked trace of traverseBackward()
3. **Insert / delete at beginning, end, and a position**
   Why the tail pointer makes end operations O(1)
4. **Singly vs. doubly: rewiring cost**
   2 pointer assignments vs. 4, made concrete
5. **Pitfalls, and a mini browser history**
   A worked application built entirely on prev/next

CORE IDEA

The Doubly Linked List Concept

Every node stores three things: data, a pointer to next, and a pointer to prev. The list keeps both head and tail — the reason end-insertion becomes O(1).



teal = next · orange = prev

CORE IDEA

Node Structure: prev, data, next

doubly_linked_list.cpp

```
struct DNode {
    int data;
    DNode* prev;
    DNode* next;
    DNode(int value)
      : data(value), prev(nullptr), next(nullptr) {}
};
```

One extra pointer

The only structural difference from a singly linked node — but it changes every operation's cost.

Half-correct is possible

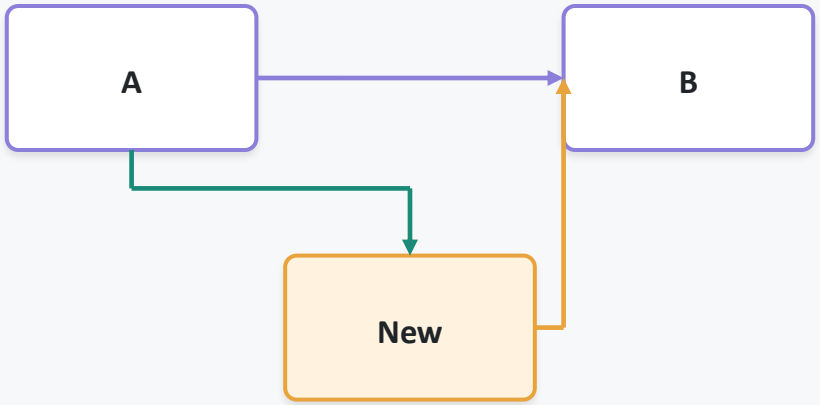
Forgetting to update prev doesn't crash — forward traversal can look perfectly fine while backward is broken.

TRADE-OFF

Singly vs. Doubly: Rewiring a Middle Insertion

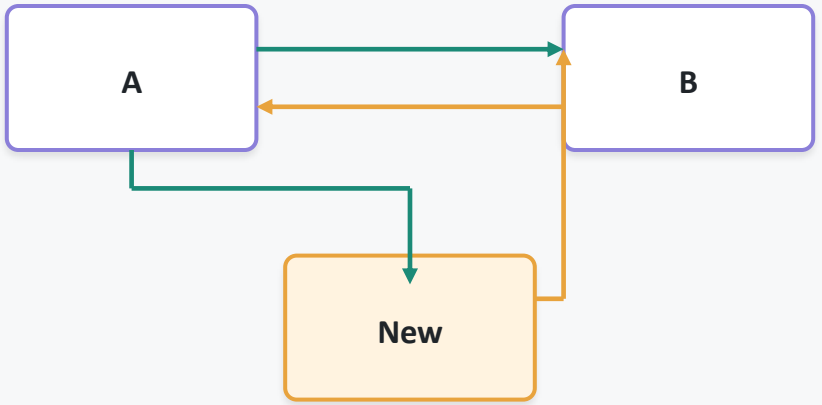
Inserting a new node between A and B takes 2 pointer assignments in a singly linked list, but 4 in a doubly linked list.

Singly — 2 assignments



1. New->next=A->next 2. A->next=New

Doubly — 4 assignments



1-2. New->next/prev 3-4. A->next=New, B->prev=New

The DoublyLinkedList Class

doubly_linked_list.cpp

```
void insertAtBeginning(int value) {
    DNode* newNode = new DNode(value);
    if (head == nullptr) {
        head = tail = newNode; return;
    }
    newNode->next = head;
    head->prev = newNode;
    head = newNode;
}
void insertAtEnd(int value) {  // O(1): tail known
    newNode->prev = tail;
    tail->next = newNode;
    tail = newNode;
}
```

insertAtBeginning

Same shape as singly linked, plus one extra line: head->prev = newNode.

insertAtEnd — no walk needed

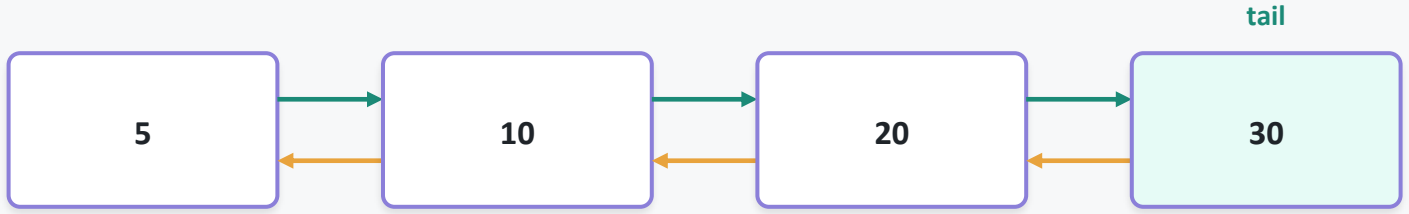
tail already points at the last node, so this is O(1) instead of the singly list's O(n).

O(1)

Both beginning and end insertion — the direct payoff of the tail pointer

Worked Trace: traverseBackward()

Starts at tail instead of head, and follows prev instead of next — otherwise the exact same loop shape.



Step	current	Printed	Next current
1	30 (tail)	30	current->prev -> 20
2	20	20	current->prev -> 10
3	10	10	current->prev -> 5
4	5	5	current->prev is nullptr -> stop

Common Pitfalls With Doubly Linked Lists

Forgetting prev (or next)

Every non-end insert/delete touches 4 pointers, not 2 — missing one leaves the structure "half correct."

Forgetting to update tail

deleteFromEnd must set tail = tail->prev; if that's nullptr, head must reset too.

Off-by-one at the last node

insertAtPosition/deleteAtPosition special-case current == tail to avoid dereferencing nullptr.

Assuming bugs crash immediately

A wrong prev only breaks backward traversal — forward, search, updateAt never notice.

Application: A Mini Browser History

back()/forward() map directly onto prev/next — both O(1) precisely because every node already stores both directions.

browser_history.cpp

```
class BrowserHistory {
private:
    PageNode* current;
public:
    void back() {
        if (current->prev) current = current->prev;
    }
    void forward() {
        if (current->next) current = current->next;
    }
};
```

```
$ ./browser_history
After visiting 3 pages: mail.com
After two back(): news.com
After one forward(): docs.com
After visiting shop.com: shop.com
```

Visiting a new page from the middle of history discards everything "forward" of it — current->next = newPage drops the old chain, exactly like a real browser.

WHY IT MATTERS

Real Applications of Doubly Linked Lists

Browser history

Back and forward buttons move in both directions through visited pages.

Music / video playlists

"Previous track" / "next track" map one-to-one onto prev and next.

Undo / redo stacks

Undo walks backward through past states, redo walks forward again.

LRU caches

Move-to-front on access and evict-from-back are both O(1) with a DLL + hash table.

ANALYSIS

Complexity, Compared to a Singly Linked List

Operation	Singly	Doubly
Insert/delete at beginning	O(1)	O(1)
Insert/delete at end	O(n)	O(1)
Backward traversal	Not possible	O(n)
Extra memory / node	1 pointer	2 pointers

O(1)

End insertion — the tail pointer's direct payoff over a singly linked list

+1 ptr

Extra memory per node — the price of backward traversal

KT

Key Takeaways

- A doubly linked list's node adds a prev pointer alongside next, and the list tracks both a head and a tail.
- The tail pointer is what makes insertion and deletion at the end O(1) — fixing the singly linked list's weakest operation, at the cost of one extra pointer per node.
- A middle insertion costs 4 pointer assignments versus 2 in a singly linked list — the direct price of traversing backward.
- Backward traversal is a simple O(n) walk from tail following prev — identical in shape to forward traversal.
- A missed prev update can leave forward traversal looking completely correct while backward traversal is silently broken — test both directions.