5

Linked Lists: Fundamentals

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

Agenda / Learning Objectives

1. **Why arrays alone aren't enough**
   Fixed size and O(n) middle insertion motivate a new structure
2. **Nodes, head, and tail**
   The self-referential struct and how a list is organized in memory
3. **Traversal, insertion, deletion**
   At the front, and at an arbitrary position
4. **Edge cases and the classic bug**
   The one-node list, and deleting without saving next first
5. **Arrays vs. linked lists**
   A direct trade-off comparison to close the lecture

Why Arrays Alone Aren't Enough

Fixed size

An array's capacity is set at creation — growing it means allocating a new block and copying everything over.

O(n) middle insertion

Inserting or removing anywhere but the end shifts every later element.

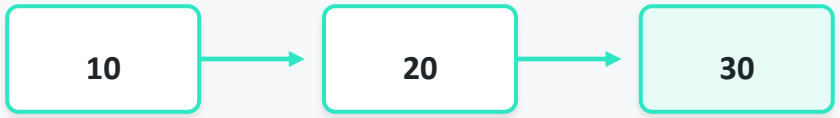
A linked list fixes both

It grows one element at a time, and insertion/deletion only changes a couple of pointers — no shifting.

Array insert at index 1 — everything shifts right

shift → shift → shift

Linked list insert — only 2 pointers change

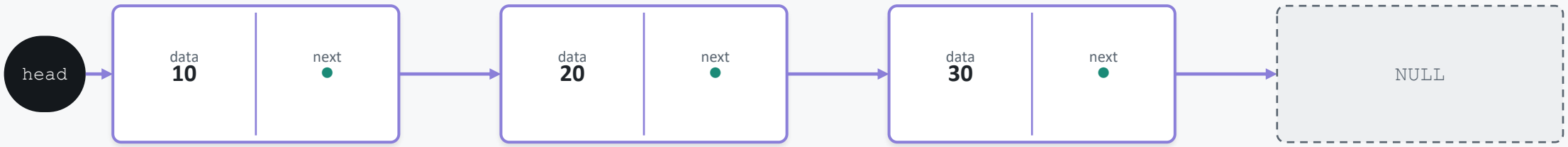


new node — 2 pointer writes only

CORE IDEA

The Linked List Concept & Memory Layout

Each node is its own independently-allocated chunk of memory; the chain of pointers is what makes it a "list" — nodes can be scattered anywhere.



Nodes may live at completely different, non-contiguous addresses in memory.

CORE IDEA

The Node Structure

A node bundles the data with a pointer to the next node — a self-referential structure.

linked_list_basics.cpp

```
struct Node {
    int data;       // the value this node holds
    Node* next;     // address of the next node,
                    // or nullptr if this is last
};
```

Self-referential

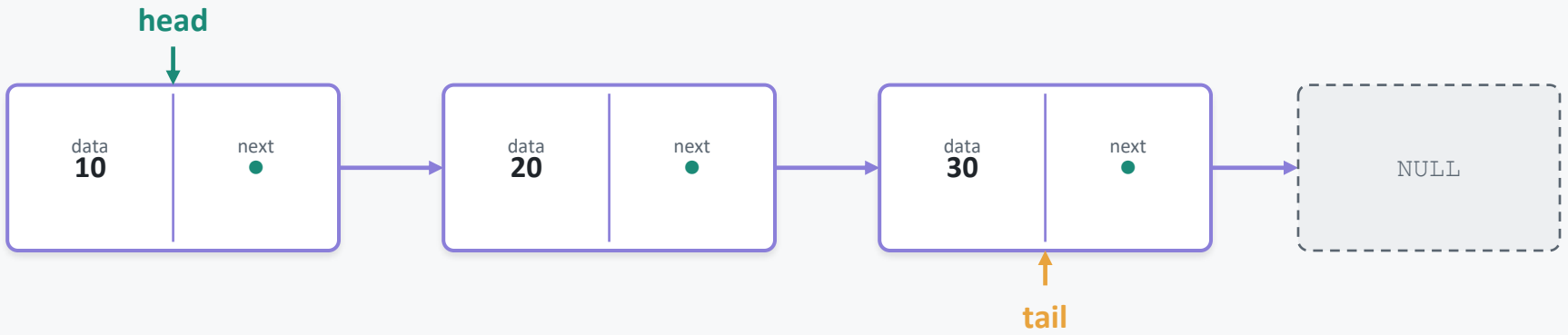
A Node contains a pointer to another Node of the exact same type.

The one idea behind everything

This single pattern makes linked lists — and later, trees and graphs — possible.

CORE IDEA

Head and Tail



head

Pointer to the first node — the only thing needed to reach the entire list. nullptr means the list is empty.

tail

The last node — the one whose next is nullptr, marking the end of the chain.

Creating and Traversing a List

linked_list_basics.cpp

```
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
    return newNode;
}

void traverse(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data;
        current = current->next;
    }
}
```

```
$ ./linked_list_basics
List: 10 -> 20 -> 30
```

createNode(value)

Allocates one node on the heap, sets its data, and nulls its next pointer.

traverse(head)

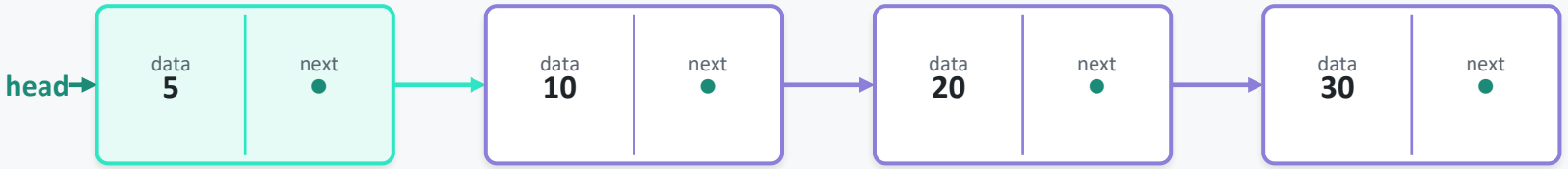
Walks from head until current is nullptr — the standard O(n) walk pattern.

Insertion at the Front

Create a new node, point its next at the current head, then make the new node the head. No existing node moves.

BEFORE

AFTER insertAtFront(head, 5)



1. new node's next = old head 2. head = new node

Insertion & Deletion at the Front — Code

linked_list_insert.cpp

```cpp
// Insert value at the front; returns the new head.
Node* insertAtFront(Node* head, int value) {
    Node* newNode = createNode(value);
    newNode->next = head;
    return newNode;
}
```

linked_list_delete.cpp

```cpp
// Delete the front node; returns the new head.
Node* deleteFromFront(Node* head) {
    if (head == nullptr) return nullptr;
    Node* oldHead = head;
    head = head->next;   // save first!
    delete oldHead;
    return head;
}
```

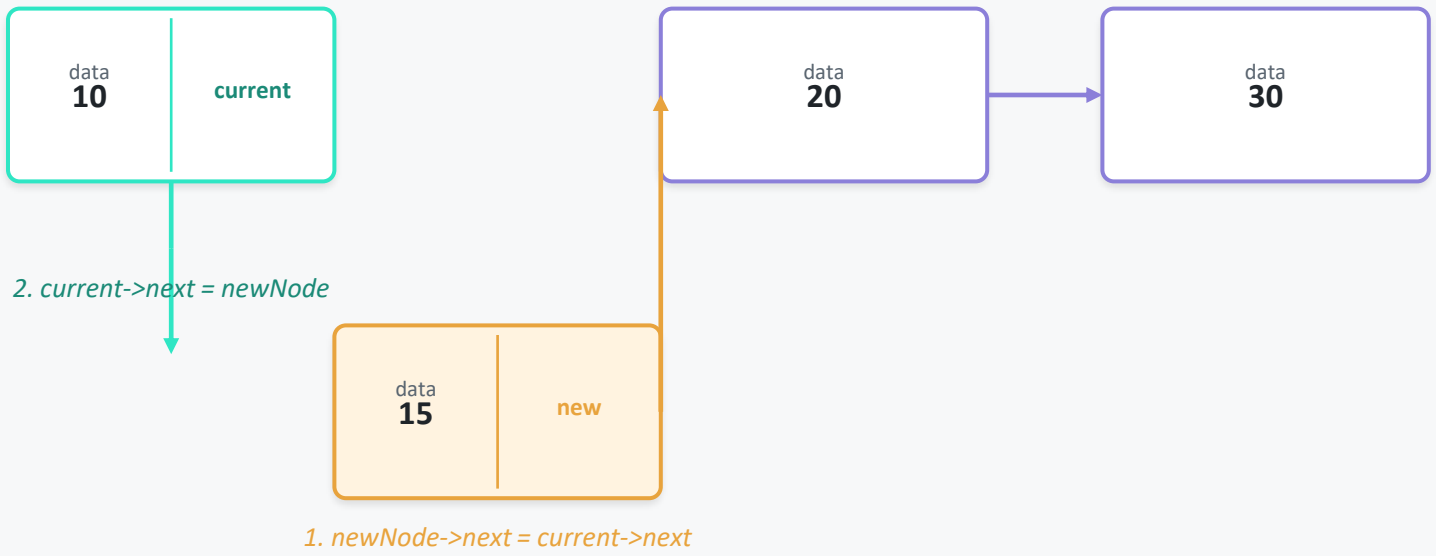
O(1)

Both operations — only pointer assignments happen, no other node moves

Always update the pointer BEFORE calling delete — reading a freed node's fields is undefined behavior.

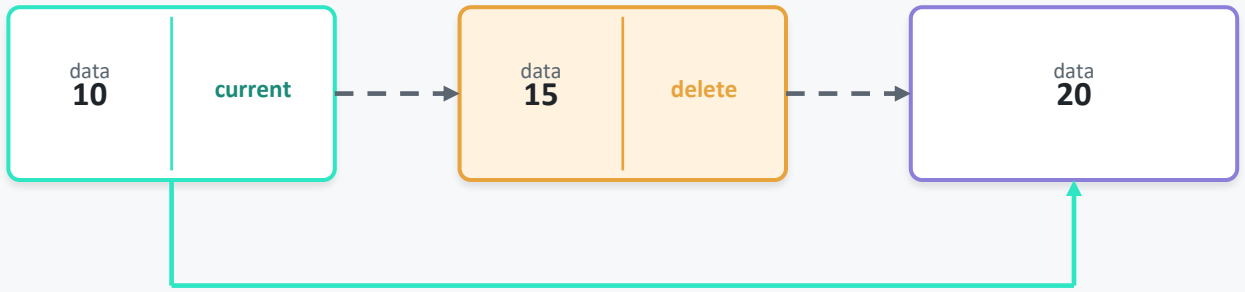
Insertion at an Arbitrary Position

Walk from head until you're standing on the node just before the target position, then apply the same two-pointer trick, anchored there.



Deletion at an Arbitrary Position

Walk to the node just before the target, remember the node to delete, splice it out, then free it.



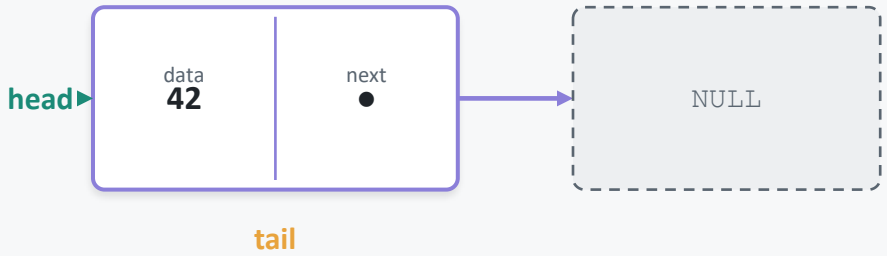
current->next = toDelete->next (save this link BEFORE deleting toDelete)

O(k)

Walking k nodes in from the head to find the splice point

Edge Case: The One-Node List

Here, "beginning" and "end" collide — head and the only node's next == nullptr describe the same node. Off-by-one bugs like to hide exactly here.



Delete the only node

head->next is nullptr, so head correctly becomes nullptr — an empty list, not a dangling pointer.

Insert into an empty list

newNode->next = head (which is nullptr) rebuilds a valid one-node list automatically.

Not a special branch

Code that checks nullptr correctly everywhere handles this case for free.

The Classic Bug: Delete Without Saving next

✗ **Broken order — undefined behavior**

```
delete head;          // memory freed
head = head->next;    // reads FREED memory!
```

✓ **Correct order**

```
Node* oldHead = head;    // still valid
head = head->next;       // read BEFORE freeing
delete oldHead;          // safe now
```

In the broken order, freed memory might still "look" correct — many systems don't overwrite it immediately. That's what makes this bug dangerous: it can pass testing and fail unpredictably once that memory is reused.

Advantages & Limitations

Advantages

Genuinely dynamic size, no wasted pre-allocation. Front insert/delete never shifts other elements.

Limitations

No random access (O(n) walk to reach node[i]); extra memory per node; worse cache locality than an array.

	Array	Linked List
Access by index	O(1)	O(n)
Insert/delete at front	O(n)	O(1)
Extra mem / element	None	One pointer

O(1)

Front insert / delete

O(n)

Indexed access

Lecture 6 builds the full toolkit: insert/delete at the end, at a position, searching, and reversing.

KT

Key Takeaways

- A linked list stores each element in its own node, scattered anywhere in memory, connected by next pointers — contiguity is traded for dynamic size.
- A node is self-referential: it holds data plus a pointer to another node of the same type.
- Head is the entry point to the list; a nullptr next marks the tail.
- Front insert/delete is O(1) — only pointer assignments happen — but reaching any position needs an O(n) walk from head.
- Always save the pointer you'll need before calling delete — reading a freed node's field is undefined behavior.