6

Singly Linked List Operations

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

Agenda / Learning Objectives

1. **The SinglyLinkedList class**
   Wrapping node-pointer logic into one reusable, self-contained ADT

2. **Insert / delete at end and at a position**
   The O(n) and O(k) operations Lecture 5 didn't cover

3. **Delete by value, search, update, count**
   The rest of the operation toolkit

4. **Reversing a list in place**
   The classic three-pointer, single-pass algorithm

5. **Complexity of every operation**
   A full table, compared directly against arrays

The Singly Linked List Class

Each node points only forward — there is no way to go backward. Wrapping the pointer logic in a class keeps head private and turns every operation into a method.

singly_linked_list.cpp

```
class SinglyLinkedList {
private:
    Node* head;
public:
    SinglyLinkedList() : head(nullptr) {}
    void insertAtBeginning(int value);
    void insertAtEnd(int value);
    void insertAtPosition(int v, int pos);
    bool deleteByValue(int value);
    int  search(int value) const;
    void reverse();
    // ...
};
```

Private head

The caller never touches Node or head directly — only the class's public methods.

One class, one program

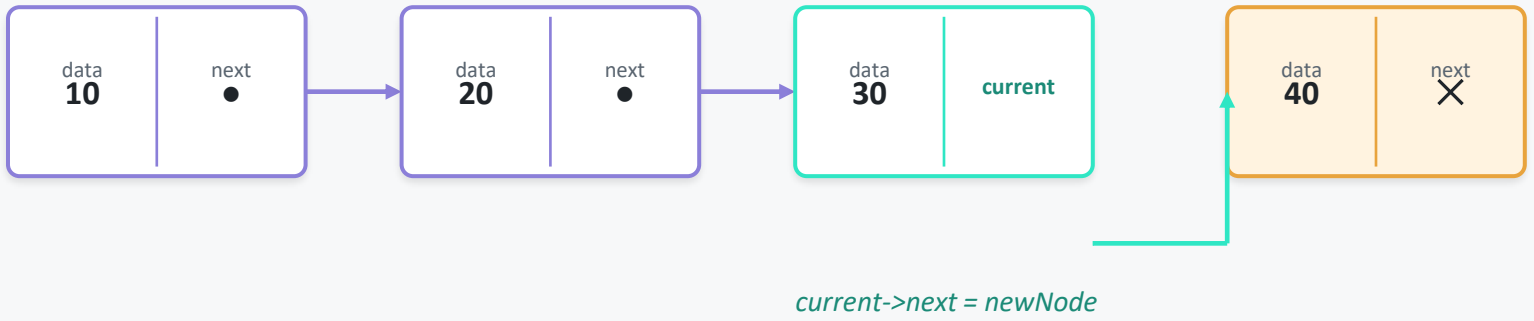
Every operation below is a method on this same class, compiled and run together.

Builds on Lecture 5

insertAtBeginning / deleteFromBeginning are the same O(1) logic, just as methods.

insertAtEnd: Walk, Then Attach

There is no tail pointer to shortcut with — must walk every node until current->next is nullptr. This walk is the entire reason the operation is O(n).

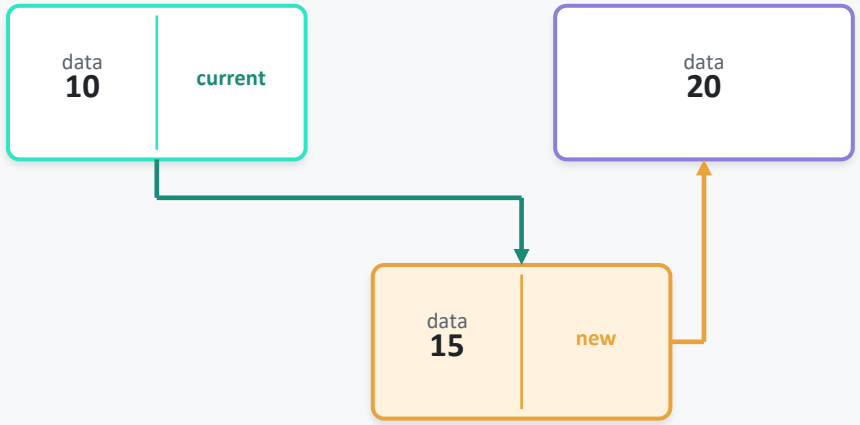


O(n)

No shortcut — Lecture 7's doubly linked list adds a tail pointer to fix this

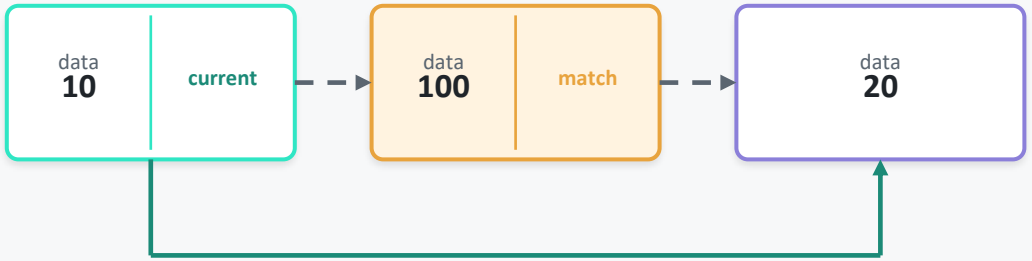
insertAtPosition & deleteByValue

insertAtPosition — splice in the middle (same pattern as Lecture 5)



1. new→next=cur→next 2. cur→next=new

deleteByValue — search, then splice out



current->next = toDelete->next

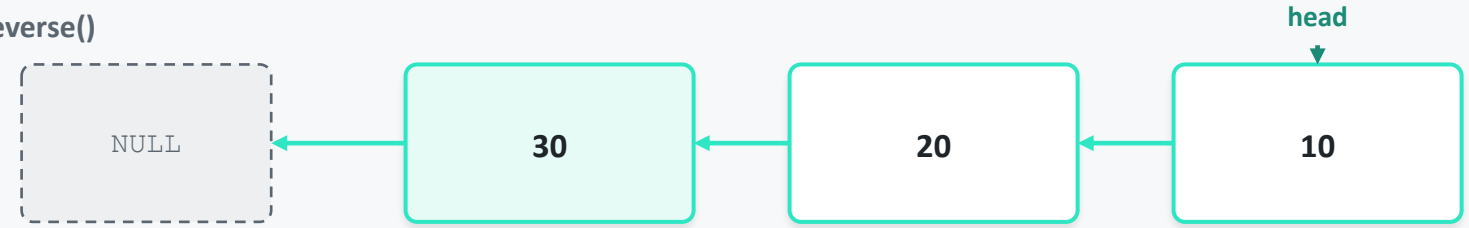
Walks like search (checks current->next->data), then splices like deleteAtPosition.

reverse(): Every next Pointer Flips

One pass, three pointers (previous, current, nextNode) — every next pointer ends up pointing backward, and head moves to the old tail.

BEFORE reverse()

AFTER reverse()



reverse() — Code

singly_linked_list.cpp

```
void reverse() {
    Node* previous = nullptr;
    Node* current = head;
    while (current != nullptr) {
        Node* nextNode = current->next;
        current->next = previous;
        previous = current;
        current = nextNode;
    }
    head = previous;
}
```

nextNode saved first

Saved before current->next is overwritten — otherwise the rest of the list is lost.

current->next = previous

This is the actual flip — the pointer now points backward instead of forward.

head = previous

previous finishes the walk sitting on the old last node — the new head.

O(n), one pass

No new nodes allocated, no data copied — only pointers move

Search, Update, and Count

search(value)

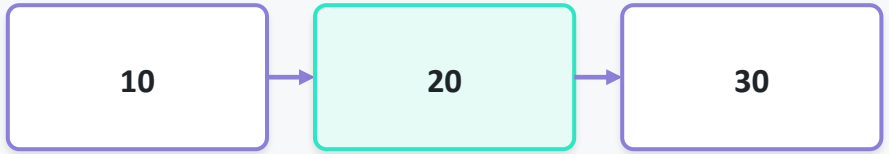
Compares data against every node from head, in order, until a match or the end. No shortcut — the same "no random access" limitation from Lecture 5.

updateAt(position, newValue)

Walks position nodes in from head, then overwrites data directly.

countNodes()

Visits every node once, incrementing a counter — no length field is stored anywhere.



search(20) walks index 0 → 1, finds a match, returns 1.

Operation	Complexity
search	O(n)
updateAt(k)	O(k)
countNodes	O(n)

ANALYSIS

Complexity of Singly Linked List Operations

Operation	Complexity
Insert at beginning	O(1)
Insert at end	O(n)
Insert at position k	O(k)
Delete from beginning	O(1)
Delete from end	O(n)
Delete at position k	O(k)
Delete by value	O(n)
Search	O(n)
Reverse	O(n)

O(1)

Only front insert / delete — everything else needs a walk

no tail

Why insertAtEnd is O(n) here — Lecture 7 fixes this

A singly linked list flips the array's trade-off: cheap front insert/delete, expensive access by position.

Common Pitfalls

Losing the rest of the list

Overwriting next before saving what it pointed to silently detaches every later node — a memory leak with no crash.

Off-by-one position indices

insertAtPosition walks to position-1, not position — the single most common typo in this code.

Forgetting the empty-list check

Skip "is head valid?" and the first call on a fresh list dereferences nullptr.

Dereferencing after delete

Reading any field of a node after delete-ing it is undefined behavior, not just risky.

Key Takeaways

- Wrapping node-pointer logic inside a class turns loose functions into a reusable, self-contained ADT — the caller never touches Node or head directly.
- Insert/delete at the beginning are O(1); insert/delete at the end or a specific position are O(n) or O(k) — reaching that point requires walking the list.
- Deleting by value combines a search-style walk with a deleteAtPosition-style splice — O(n) because the position isn't known ahead of time.
- Reversing is a classic O(n), single-pass algorithm using three pointers to flip every next pointer — the old tail becomes the new head.
- Every operation's complexity comes down to one fact: a singly linked list can only be walked forward, one node at a time, from the head.