Linked Lists as an Abstract Data Type

Data Structures & Algorithms

Class 02

Topic: ADT, Visual Operations, Singly & Doubly Linked Lists

Pranshu Mandal

Learning Objectives

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

1. Understand what an Abstract Data Type (ADT) is
2. Distinguish between ADT specification and implementation
3. Visualize how linked list operations work step-by-step
4. Compare singly and doubly linked lists
5. Choose the right linked list type for specific use cases
6. Understand when to use linked lists vs arrays

Recap: Class 01

In our last class, we learned:

- A linked list is a chain of nodes connected via pointers
- Each node contains data and a reference to the next node
- Unlike arrays, linked lists don't require contiguous memory
- Basic node structure and creation in C

Today: Abstract Data Types and different types of linked lists!

Understanding Abstract Data Types (ADT)

What is an ADT?

An **Abstract Data Type (ADT)** separates **WHAT** you can do from **HOW** it's done.

ADT defines:

1. **Data:** What information is stored?
2. **Operations:** What can you do with the data?
3. **Behavior:** How should each operation behave?
4. **NOT included:** How it's stored in memory!

Key Point: Interface vs Implementation

Analogy 1: Bank Account

When you have a bank account, you can:

- **Deposit** money
- **Withdraw** money
- **Check balance**
- **Transfer** to another account

Do you know HOW the bank stores your money?

- Database type? Encryption method? Interest calculation?

You don't need to know! That's an ADT - you use the interface!

Analogy 2: Phone Contacts

Your phone's contact list lets you:

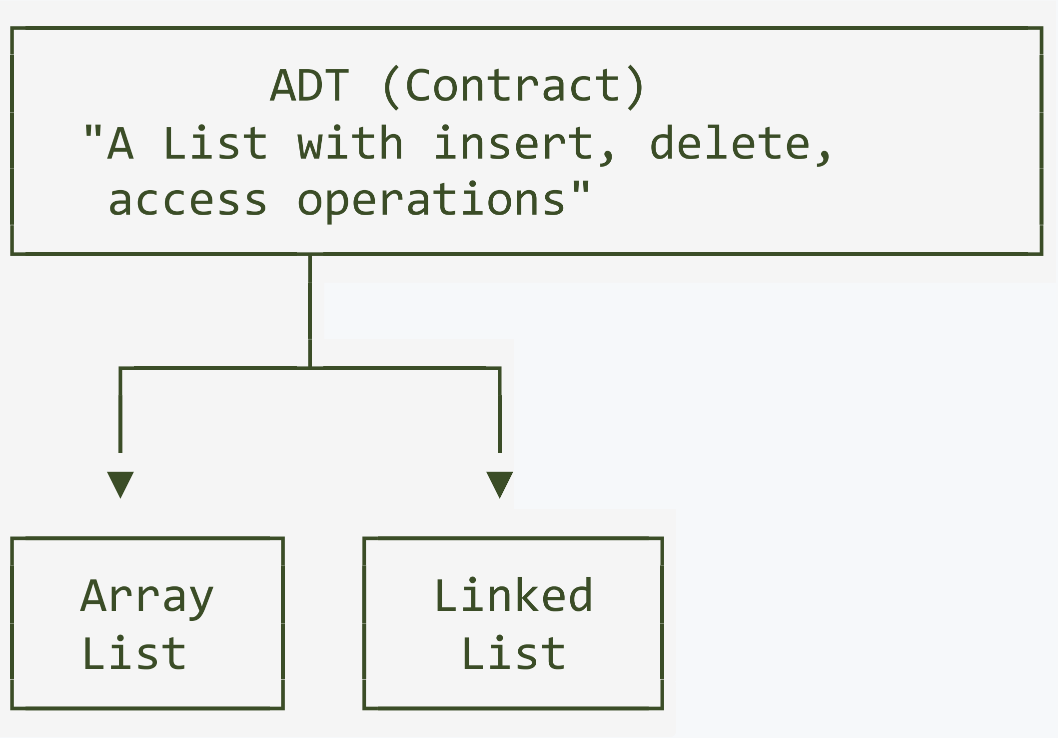
- **Add** a new contact
- **Delete** a contact
- **Search** for a contact
- **Update** contact info

Different phones store contacts differently:

- Apple: sorted array
- Android: hash table
- Old Nokia: linked list

You don't care! As long as "Add Contact" works!

ADT vs. Implementation



Same interface, different performance!

Real-World ADT Examples

ADT	Can be implemented as...
List	Array, Linked List, Skip List
Stack	Array, Linked List
Queue	Array (circular), Linked List
Set	Hash Table, Binary Search Tree
Map	Hash Table, Red-Black Tree

The benefit: Swap implementations without changing your program!

ADT Operation Categories

1. Creators - Make new instances

```
createEmptyList(), createStack()
```

2. Mutators - Change the ADT's state

```
insert(list, element, position)
delete(list, position)
```

3. Observers - Read state without changing

```
get(list, position), size(list), isEmpty(list)
```

4. Producers - Create new instances from existing

```
concat(list1, list2), reverse(list)
```

Linked List ADT Operations

Category	Operations
Main	Insert, Delete, Access
Auxiliary	Count, Search, isEmpty, Clear

The ADT tells us WHAT these do, not HOW they work!

Now let's see HOW with visuals...

Linked List Operations - Visual Journey

Setup: Our Node Structure

Each node has two parts:

```
┌──────────┬──────────┐
│   DATA   │   NEXT   │
└──────────┴──────────┘
   Value    Pointer to next
```

Example list:

```
HEAD
  ↓
┌────┬────┐   ┌────┬────┐   ┌────┬────┐   ┌────┬──────┐
│ 10 │  •─┼──→│ 20 │  •─┼──→│ 30 │  •─┼──→│ 40 │ NULL │
└────┴────┘   └────┴────┘   └────┴────┘   └────┴──────┘
```

Insert at Beginning - Step 1

Goal: Insert value `5` at the start

Starting point:

```
HEAD
  ↓
┌────┬───┐   ┌────┬───┐   ┌────┬──────┐
│ 10 │ •─┼──→│ 20 │ •─┼──→│ 30 │ NULL │
└────┴───┘   └────┴───┘   └────┴──────┘
```

Step 1: Create new node

```
         ┌────┬──────┐
newNode: │  5 │ NULL │
         └────┴──────┘
```

Insert at Beginning - Step 2

Step 2: Point new node to current HEAD

```
HEAD
  ↓
┌────┬───┐  ┌────┬───┐  ┌────┬──────┐
│ 10 │ •─┼─→│ 20 │ •─┼─→│ 30 │ NULL │
└────┴───┘  └────┴───┘  └────┴──────┘
  ↑
  │
┌────┬───┐
│  5 │ • │  newNode->next = HEAD
└────┴───┘
```

Insert at Beginning - Complete

Step 3: Update HEAD to point to new node

```
         HEAD
          ↓
Result: ┌────┬───┐   ┌────┬───┐   ┌────┬───┐   ┌────┬──────┐
        │  5 │ •─┼──→│ 10 │ •─┼──→│ 20 │ •─┼──→│ 30 │ NULL │
        └────┴───┘   └────┴───┘   └────┴───┘   └────┴──────┘
```

Time Complexity: O(1) (Constant Time)

- Only 2 pointer updates, regardless of list size!

Common Mistake: Update HEAD before connecting to old list = lose the chain!

Insert at End - O(n)

Goal: Insert value `50` at the end

Step 1: Traverse to find the last node

```
current              current        current
  ↓                    ↓              ↓
┌────┬───┐  ┌────┬───┐  ┌────┬──────┐
│ 10 │ •─┼─→│ 20 │ •─┼─→│ 30 │ NULL │  ← Found!
└────┴───┘  └────┴───┘  └────┴──────┘
```

Step 2: Link last node to new node

```
┌────┬───┐  ┌────┬───┐  ┌────┬───┐  ┌────┬──────┐
│ 10 │ •─┼─→│ 20 │ •─┼─→│ 30 │ •─┼─→│ 50 │ NULL │
└────┴───┘  └────┴───┘  └────┴───┘  └────┴──────┘
```

Time Complexity: O(n) (Linear Time)

Optimization: Tail Pointer

With tail pointer: Insert at end becomes O(1)!

```
HEAD                                       TAIL
  ↓                                          ↓
┌────┬───┐   ┌────┬───┐   ┌────┬──────┐
│ 10 │ •─┼──→│ 20 │ •─┼──→│ 30 │ NULL │
└────┴───┘   └────┴───┘   └────┴──────┘

Insert steps:
1. newNode->next = NULL
2. TAIL->next = newNode
3. TAIL = newNode
```

Only 3 pointer updates!

Delete from Beginning

Goal: Remove first element

Step 1 & 2: Save reference, move HEAD

```
temp = HEAD
  ↓
┌────┬───┐   ┌────┬───┐   ┌────┬──────┐
│ 10 │ • │   │ 20 │ •───→ 30 │ NULL │
└────┴───┘   └────┴───┘   └────┴──────┘
                ↑
             HEAD (updated)
```

Step 3: Free the old first node

```
          HEAD
            ↓
         ┌────┬───┐   ┌────┬──────┐
Result:  │ 20 │ •───→ 30 │ NULL │
```

Count Nodes - Visual Animation

Goal: Count how many nodes

```
Step 0: count = 0
Step 1: count = 1, visit [10]
Step 2: count = 2, visit [20]
Step 3: count = 3, visit [30]
Step 4: current == NULL, stop!

Return count = 3
```

Time Complexity: O(n) - Visit every node exactly once

Why O(n)? Must visit every node to count

Compare with arrays: Array length: O(1) | Linked list: O(n)

Find nth from End - Two-Pointer Technique

Find 2nd from end:

```
Step 1: Move fast 2 steps ahead
slow        fast
  ↓           ↓
[10] → [20] → [30] → [40] → [50] → NULL

Step 2: Move both until fast reaches NULL
                slow        fast
                  ↓           ↓
[10] → [20] → [30] → [40] → [50] → NULL
                                      ↓
                                    NULL
```

Result: slow points to `40` - the 2nd from end!

Time Complexity: O(n) - Single pass!

Type of Linked List: Singly Linked Lists

Structure and Characteristics

Singly Linked List: Each node has:

- **Data** field
- **One pointer** to the next node

Visual:

```
HEAD → [Data|Next] → [Data|Next] → [Data|Next] → [Data|NULL]

Memory View:
Node 1: Addr 1000 → [10 | 2500]
Node 2: Addr 2500 → [20 | 7800]
Node 3: Addr 7800 → [30 | 1200]
Node 4: Addr 1200 → [40 | NULL]
```

Key: Can ONLY move forward! No going back!

Advantages of Singly Linked Lists

Advantage	Explanation
Simple Structure	Only one pointer per node
Memory Efficient	Minimal overhead
Fast Head Operations	O(1) insert/delete at beginning
Dynamic Growth	No pre-allocation needed

Real-World: Like a treasure hunt - each clue points to next, can't go back!

Disadvantages of Singly Linked Lists

Disadvantage	Impact
No Backward Traversal	Can't move to previous node
Sequential Access Only	Must start from HEAD
Deletion Needs Previous	O(n) to delete
No Direct Tail Access	Without tail pointer, O(n) to reach end

Problem: Even if you have a pointer to node, need previous node to delete it!

Use Case 1: Implementing a Stack

```
STACK (LIFO - Last In, First Out)
TOP
 ↓
┌────┬───┐   ┌────┬───┐   ┌────┬──────┐
│  3 │ •─┼──→│  2 │ •─┼──→│  1 │ NULL │
└────┴───┘   └────┴───┘   └────┴──────┘
                            ↑ Bottom

push(4) → Insert at HEAD (O(1))
pop()   → Delete from HEAD (O(1))
```

Perfect for:

- Function call stack
- Undo functionality
- Expression evaluation

Use Case 2: Queue (with Tail Pointer)

```
QUEUE (FIFO - First In, First Out)
HEAD                                      TAIL
 ↓                                         ↓
┌─────┬─────┐   ┌─────┬─────┐   ┌─────┬───────┐
│  1  │  •──┼──→│  2  │  •──┼──→│  3  │ NULL  │
└─────┴─────┘   └─────┴─────┘   └─────┴───────┘
  ↑ Remove                        ↑ Add

enqueue(4) → Insert at TAIL (O(1))
dequeue()  → Delete from HEAD (O(1))
```

Perfect for:

- Print queue, Task scheduling, Buffer management

Another Type: Doubly Linked Lists

Structure and Characteristics

Doubly Linked List: Nodes with:

- **Data** field
- **Two pointers:** `next` AND `prev`

Visual:

```
NULL ← [Prev|Data|Next] ↔️ [Prev|Data|Next] ↔️ [Prev|Data|Next] → NULL
         ↑                                                   ↑
        HEAD                                                TAIL

Bidirectional Navigation:
Forward:  HEAD → 10 → 20 → 30 → NULL
Backward: NULL ← 10 ← 20 ← 30 ← TAIL
```

Advantages of Doubly Linked Lists

Advantage	Impact
Bidirectional Traversal	Navigate in both directions
Easier Deletion	Have access to previous node
Efficient Tail Operations	O(1) delete from end with tail pointer
Better for Algorithms	Many algorithms need prev access

Real-World: Like a two-way street - can go back if you miss your turn!

Disadvantages of Doubly Linked Lists

Disadvantage	Impact
More Memory	2 pointers instead of 1
Complex Management	Must update 2 pointers on changes
Larger Nodes	Fewer nodes fit in cache

Memory Cost:

```
Singly: 12 bytes per node
Doubly: 20 bytes per node

For 1000 nodes:
Extra cost: 8,000 bytes = 67% more!
```

Operations Comparison

Operation	Singly LL	Doubly LL	Winner
Insert at Head	O(1), 2 updates	O(1), 4 updates	Singly
Delete from Head	O(1)	O(1)	Tie
Insert at Tail (w/ tail)	O(1), 2 updates	O(1), 4 updates	Tie
Delete from Tail (w/ tail)	O(n)	O(1)	Doubly
Reverse Traversal	Impossible	O(n)	Doubly
Memory per Node	12 bytes	20 bytes	Singly

Why Doubly Wins at Tail Delete

Singly LL: O(n) even with TAIL!

- Need to find second-to-last node
- Must traverse from HEAD

Doubly LL: O(1) with TAIL!

```
TAIL->prev->next = NULL  // Previous node's next = NULL
TAIL = TAIL->prev         // Update TAIL
Free old tail
```

Direct access to previous via prev pointer!

Real-World: Browser History

```
Browser Navigation:

NULL ← [google.com] ↔️ [youtube.com] ↔️ [github.com] → NULL
          ↑                              ↑
       3 pages back                    Current

Back button:    current = current->prev
Forward button: current = current->next
```

Why doubly? Need to go both directions!

Real-World: Music Player

```
Playlist:

NULL ← [Song 1] ↔️ [Song 2] ↔️ [Song 3] ↔️ [Song 4] → NULL
                    ↑
               Now playing

Previous button: current = current->prev
Next button:     current = current->next
```

Perfect for: Previous/next song controls!

Real-World: LRU Cache

```
LRU Cache with 3 slots:

Most Recent                    Least Recent
    ↓                               ↓
NULL ← [Page C] ↔ [Page A] ↔ [Page B] → NULL
         ↑                  ↑
        HEAD               TAIL

Access Page A: Move to front (most recent)
Cache full? Remove from tail (least recent)
```

Why doubly? Need to remove from middle and add to front!

When to Use Each Type

Decision Matrix

```
Choose SINGLY LINKED LIST when:
  ✅ Only need forward traversal
  ✅ Memory is limited/critical
  ✅ Implementing Stack (LIFO)
  ✅ Implementing Queue (with tail pointer)
  ✅ Simple list with rare deletions

Choose DOUBLY LINKED LIST when:
  ✅ Need bidirectional traversal
  ✅ Frequent deletions from end
  ✅ Need to delete with only node reference
  ✅ Implementing Deque (double-ended queue)
  ✅ Browser history, undo/redo, LRU cache
```

Case Study 1: Call Stack

Requirements:

- Functions are called (push) and return (pop)
- Only need most recent function
- LIFO behavior
- No backward traversal needed

Decision: Singly Linked List

Why?

- Only insert/delete from HEAD (O(1))
- No backward traversal needed
- Memory efficient

Case Study 2: Music Player App

Requirements:

- Users click "Previous" and "Next"
- Need to remember playback position
- Playlist can be edited

Decision: Doubly Linked List

Why?

- Bidirectional navigation needed
- Easy deletion from playlist
- Can jump backward easily

Quick Arrays vs Linked Lists

Comparison Table

Criterion	Array	Linked List	Best Choice
Random Access	O(1)	O(n)	Array
Insert at Start	O(n)	O(1)	Linked List
Insert at End	O(1)*	O(n) or O(1)†	Depends
Delete from Start	O(n)	O(1)	Linked List
Memory	Contiguous	Scattered	Array (cache)
Size	Fixed/resizable	Dynamic	Linked List

*For dynamic arrays

†With tail pointer

When to Use What?

Use ARRAY when:

- Need fast random access
- Know size in advance
- Mostly reading, rare inserts/deletes
- Cache performance is critical

Examples: Image data, lookup tables, game grids

Use LINKED LIST when:

- Frequent inserts/deletes at start
- Size unknown and highly dynamic
- Can't allocate large contiguous memory
- Implementing Stack, Queue, Deque

Summary

Key Takeaways

ADT (Abstract Data Type) separates WHAT from HOW

- Interface vs Implementation
- Same ADT, multiple implementations

Visual operations show pointer manipulation

- Order of pointer updates matters
- Always connect before disconnecting

Singly LL: Simple, memory-efficient, forward-only

- Perfect for Stack, Queue

Doubly LL: Bidirectional, easier deletion

- Perfect for browser history, music player, LRU cache