Linear Data Structures

Minseok Jeon
DGIST

November 2, 2025

Contents

Introduction

What Are Linear Data Structures?

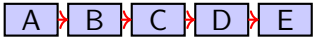
Definition

Linear data structures are structures where elements are arranged **sequentially**, with each element having at most one predecessor and one successor.

Covered Structures:

- Arrays (Static & Dynamic)
- Linked Lists (Singly, Doubly, Circular)
- Stacks (LIFO)
- Queues (FIFO)
- Deques & Circular Queues
- Priority Queues

Linear Arrangement



Sequential Order

Key Characteristic

Arrays

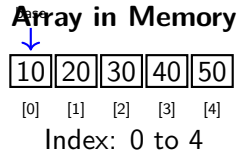
Arrays: The Foundation

Definition

An **array** is a contiguous block of memory storing elements of the same type in sequential memory locations, providing **O(1) indexing**.

Key Properties:

- Contiguous memory allocation
- Direct access by index
- Fixed or dynamic size
- Cache-friendly (spatial locality)



Memory Address

```
addr[i] = base + i × sizeof(element)
```

Static vs Dynamic Arrays

Static Arrays

Fixed size at creation

Pros:

- ✓ No resize overhead
- ✓ Predictable memory
- ✓ Slightly faster

Cons:

- × Cannot grow/shrink
- × Must know size
- × Wasted or insufficient space

Dynamic Arrays

Resizable as needed

Pros:

- ✓ Flexible size
- ✓ O(1) amortized append
- ✓ No size constraints

Cons:

- × Occasional resize (O(n))
- × Extra capacity overhead
- × Unpredictable resizing

Examples

Examples

Dynamic Array Growth Strategy

How It Works:

1. Start with small capacity (e.g., 4)
2. When full, allocate new array (2× size)
3. Copy all elements to new array
4. Free old array

Growth Sequence

Capacity: $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \ldots$

Amortized O(1)

For n insertions:

Total copies $= 1 + 2 + 4 + \ldots + n = 2n - 1$

Average: $(2n - 1)/n \approx 2 = O(1)$

Python Example

```python
class DynamicArray:
    def __init__(self):
        self._capacity = 4
        self._size = 0
        self._data = [None] * 4

    def append(self, item):
        if self._size == self._capacity:
            self._resize(2 * self._capacity)
        self._data[self._size] = item
        self._size += 1

    def _resize(self, new_cap):
        new_data = [None] * new_cap
        for i in range(self._size):
            new_data[i] = self._data[i]
        self._data = new_data
        self._capacity = new_cap
```

Array Performance Summary

Operation	Static	Dynamic	Notes
Access by index	O(1)	O(1)	Direct memory access
Search (unsorted)	O(n)	O(n)	Linear scan
Search (sorted)	O(log n)	O(log n)	Binary search
Insert at end	N/A	O(1) amortized	May trigger resize
Insert at middle	O(n)	O(n)	Shift elements
Delete	O(n)	O(n)	Shift elements
Memory usage	Exact	Extra capacity	25-50% overhead

When to Use Arrays

- Need fast random access by index
- Size is known or changes infrequently
- Mostly reading data, few insertions/deletions

Linked Lists

Introduction to Linked Lists

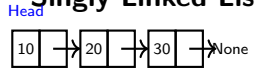
Definition

A **linked list** is a linear structure where elements (nodes) are stored non-contiguously in memory. Each node contains data and a reference (pointer) to the next node.

Key Differences from Arrays:

- × No random access (must traverse)
- ✓ Dynamic size (no preallocation)
- ✓ O(1) insert/delete at known position
- × Extra memory for pointers
- × Poor cache locality

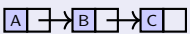
Singly Linked List



Types of Linked Lists

Singly Linked

Each node points to next only



Memory: 1 pointer/node
Traverse: Forward only

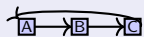
Doubly Linked

Pointers to next & prev

Memory: 2 pointers/node
Traverse: Both directions

Circular

Last points to first



Forms a circle
No NULL end

Feature	Singly	Doubly	Circular
Memory/node	1 pointer	2 pointers	1-2 pointers
Forward traverse	✓	✓	✓
Backward traverse	×	✓	× (singly)
Insert at beginning	O(1)	O(1)	O(1)
Insert at end	O(n) or O(1)*	O(1)	O(n) or O(1)*

Singly Linked List Implementation

Node Class

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Insert at End

```
def insert_at_end(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
    else:
        current = self.head
        while current.next:  # O(n)
            current = current.next
        current.next = new_node
    self.size += 1
```

Insert at Beginning

```
def insert_at_beginning(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
    self.size += 1
# O(1) - just update head
```

Delete Node

```
def delete(self, data):
    if self.head.data == data:
        self.head = self.head.next
        return True

    current = self.head
    while current.next:
        if current.next.data == data:
            current.next = current.next.next
            return True
```

Linked List vs Array

Operation	Array	Linked List
Access by index	O(1)	O(n)
Search	O(n)	O(n)
Insert at beginning	O(n)	O(1)
Insert at end	O(1) amortized	O(n) or O(1)*
Insert at middle	O(n)	O(1)**
Delete at beginning	O(n)	O(1)
Delete at middle	O(n)	O(1)**
Memory overhead	Low	High (pointers)
Cache performance	Excellent	Poor

*O(1) with tail pointer, **If position is known

Use Arrays When

- Need random access
- Size known/stable

Use Linked Lists When

- Frequent insertions/deletions
- Size unknown/dynamic

Stacks

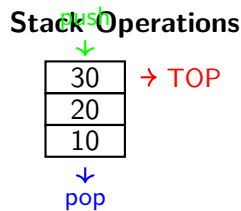
Stacks: LIFO Principle

Definition

A **stack** is a linear data structure following **LIFO** (Last In, First Out). The last element added is the first one removed, like a stack of plates.

Core Operations (All O(1)):

- `push(item)`: Add to top
- `pop()`: Remove from top
- `peek()`: View top without removing
- `is_empty()`: Check if empty
- `size()`: Get element count



Key Restriction

Only the **top** element is accessible. This restriction is intentional and enables many

Stack Implementations

Array-Based Stack

```python
class ArrayStack:
    def __init__(self):
        self._data = []

    def push(self, item):
        self._data.append(item)  # O(1)

    def pop(self):
        if self.is_empty():
            raise IndexError("empty")
        return self._data.pop()  # O(1)

    def peek(self):
        if self.is_empty():
            raise IndexError("empty")
        return self._data[-1]  # O(1)

    def is_empty(self):
        return len(self._data) == 0

    def size(self):
        return len(self._data)
```

Linked List-Based Stack

```python
class LinkedStack:
    def __init__(self):
        self._head = None
        self._size = 0

    def push(self, item):
        new_node = Node(item)
        new_node.next = self._head
        self._head = new_node
        self._size += 1  # O(1)

    def pop(self):
        if self.is_empty():
            raise IndexError("empty")
        item = self._head.data
        self._head = self._head.next
        self._size -= 1
        return item  # O(1)

    def peek(self):
        if self.is_empty():
            raise IndexError("empty")
        return self._head.data

    def is_empty(self):
        return self._head is None
```

Stack Applications

1. Function Call Stack

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

# Call stack for factorial(3):
# factorial(3)
#   factorial(2)
#     factorial(1) <- TOP
#       returns 1
#     returns 2
#   returns 6
```

2. Balanced Parentheses

```
def is_balanced(expr):
    stack = []
    pairs = {')':'(', ']':'[', '}':'{'}

    for char in expr:
        if char in pairs:
```

3. Undo/Redo

```
class TextEditor:
    def __init__(self):
        self.text = ""
        self.undo_stack = []
        self.redo_stack = []

    def write(self, text):
        self.undo_stack.append(self.text)
        self.text += text
        self.redo_stack.clear()

    def undo(self):
        if self.undo_stack:
            self.redo_stack.append(
                self.text)
            self.text = \
                self.undo_stack.pop()

    def redo(self):
        if self.redo_stack:
            self.undo_stack.append(
                self.text)
            self.text = \
                self.redo_stack.pop()
```

Queues

Queues: FIFO Principle

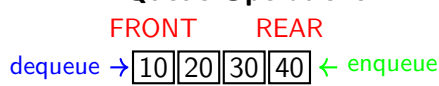
Definition

A **queue** is a linear data structure following **FIFO** (First In, First Out). The first element added is the first one removed, like a waiting line.

Core Operations (All O(1)):

- `enqueue(item)`: Add to rear
- `dequeue()`: Remove from front
- `front()`: View front without removing
- `is_empty()`: Check if empty
- `size()`: Get element count

Queue Operations



Key Property

Fair processing: first come, first served

Queue Implementations

Naive (Bad!)

```
class NaiveQueue:
    def __init__(self):
        self._data = []

    def enqueue(self, item):
        self._data.append(item)  # O(1)

    def dequeue(self):
        return self._data.pop(0)  # O(n)!
        # Shifts all elements left!
```

Using Deque (Good!)

```
from collections import deque

class Queue:
    def __init__(self):
        self._data = deque()

    def enqueue(self, item):
        self._data.append(item)  # O(1)
```

Linked List Queue

```
class LinkedQueue:
    def __init__(self):
        self._head = None
        self._tail = None
        self._size = 0

    def enqueue(self, item):
        new_node = Node(item)
        if self._tail is None:
            self._head = self._tail = \
                new_node
        else:
            self._tail.next = new_node
            self._tail = new_node
        self._size += 1  # O(1)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("empty")
        item = self._head.data
        self._head = self._head.next
        if self._head is None:
```

Queue Applications

1. Task Scheduling

```python
class TaskScheduler:
    def __init__(self):
        self.queue = Queue()

    def add_task(self, task):
        self.queue.enqueue(task)

    def process_next(self):
        if not self.queue.is_empty():
            task = self.queue.dequeue()
            print(f"Processing: {task}")

# First added = first processed
```

2. Print Queue

```python
class PrintQueue:
    def __init__(self):
        self.queue = Queue()

    def add_document(self, doc):
        self.queue.enqueue(doc)
```

3. BFS Traversal

```python
def bfs(graph, start):
    visited = set()
    queue = Queue()
    queue.enqueue(start)
    visited.add(start)

    while not queue.is_empty():
        node = queue.dequeue()
        print(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.enqueue(neighbor)

# Explores level by level
```

Other Uses

- Request handling (web servers)
- Message queues

Deque & Circular Queue

Deque (Double-Ended Queue)

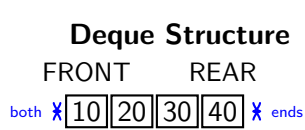
Definition

A **deque** (pronounced "deck") allows insertion and deletion at **both ends** (front and rear). It combines capabilities of stacks and queues.

Operations (All O(1)):

- `add_front(item)`
- `add_rear(item)`
- `remove_front()`
- `remove_rear()`
- `front()`, `rear()`

Deque Structure



Use Cases

- Palindrome checking
- Sliding window problems
- Browser history (fast back/forward)

Python Implementation

```
1 from collections import deque
2
3 dq = deque()
```

Circular Queue

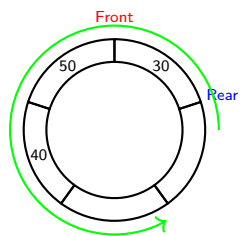
Definition

A **circular queue** reuses empty spaces when elements are dequeued. The rear wraps around to the beginning when it reaches the end.

Why Circular?

- Regular queue: empty spaces at front wasted
- Circular queue: reuses freed space
- Fixed capacity (efficient for buffers)

Circular Queue



Key Formula

```
rear = (front + size) % capacity
front = (front + 1) % capacity
```

Priority Queue

Priority Queue Overview

Definition

A **priority queue** is an ADT where each element has an associated **priority**. Elements with higher priority are dequeued first, regardless of insertion order.

Key Difference:

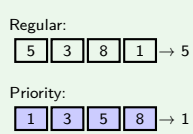
- Regular queue: FIFO
- Priority queue: Highest priority first

Operations:

- `insert(item, priority)`: O(log n)
- `extract_min/max()`: O(log n)
- `peek()`: O(1)

Visual Example

Min Priority Queue



Python Usage

```
1 import heapq
2 heap = []
```

Implementation

Priority Queue Applications

1. Emergency Room

```
class EmergencyRoom:
    def __init__(self):
        self.queue = PriorityQueue()

    def admit_patient(self, name, severity):
        # Lower = more critical
        self.queue.insert(name, severity)

    def treat_next(self):
        # Treats most critical first
        return self.queue.extract_min()

er = EmergencyRoom()
er.admit_patient("Alice", 5)  # Minor
er.admit_patient("Bob", 1)    # Critical
er.treat_next()  # Bob (critical)
```

2. Task Scheduler

```
class TaskScheduler:
    def __init__(self):
        self.pq = PriorityQueue()
```

3. Dijkstra's Algorithm

```
def dijkstra(graph, start):
    distances = {node: float('inf')
                 for node in graph}
    distances[start] = 0
    pq = PriorityQueue()
    pq.insert(start, 0)

    while not pq.is_empty():
        current = pq.extract_min()

        for neighbor, weight in \
                graph[current]:
            dist = distances[current] + \
                   weight
            if dist < distances[neighbor]:
                distances[neighbor] = dist
                pq.insert(neighbor, dist)

    return distances
```

Other Uses

- Event-driven simulation

Comparison & Use Cases

Performance Summary

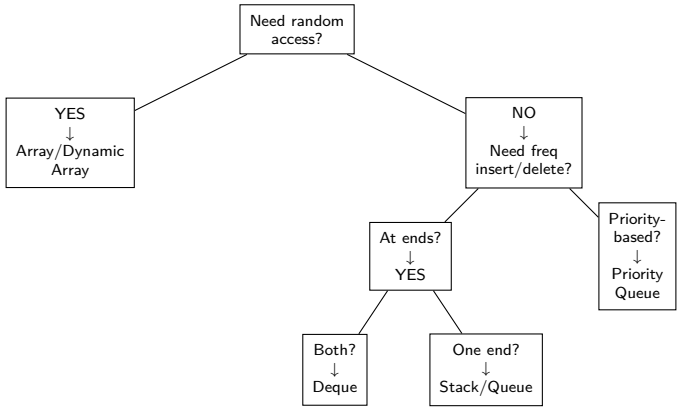
Structure	Access	Insert (begin)	Insert (end)	Delete (begin)	Delete (end)	Sear
Array	O(1)	O(n)	O(n)	O(n)	O(n)	O(n
Dynamic Array	O(1)	O(n)	O(1)*	O(n)	O(1)	O(n
Singly Linked	O(n)	O(1)	O(n)**	O(1)	O(n)	O(n
Doubly Linked	O(n)	O(1)	O(1)	O(1)	O(1)	O(n
Stack	N/A	O(1)	N/A	O(1)	N/A	N/A
Queue	N/A	N/A	O(1)	O(1)	N/A	N/A
Deque	N/A	O(1)	O(1)	O(1)	O(1)	N/A
Priority Queue	N/A	O(log n)	O(log n)	O(log n)	N/A	O(n

*amortized, **O(1) with tail pointer

Memory Overhead Comparison

- **Array**: Minimal (just elements) + potential unused capacity
- **Singly Linked List**: 1 pointer per node

Decision Guide: Which Structure to Use?



Real-World Use Cases

Scenario	Structure	Reason
Student grades (by ID)	Dynamic Array	Fast access by index, known size
Text editor undo/redo	Two Stacks	LIFO matches undo/redo behavior
Web server requests	Queue	FIFO ensures fairness
Music playlist	Doubly Linked List	Easy forward/backward, insert anywhere
CPU scheduling	Priority Queue or Circular Queue	Priority-based or round-robin
Browser history	Deque	Fast back/forward navigation
Video streaming buffer	Circular Queue	Fixed buffer, continuous

Common Mistakes to Avoid

1. Using `list.pop(0)` for Queue

```python
# BAD: O(n) - shifts all elements!
queue = []
queue.append(1)
queue.pop(0)

# GOOD: O(1) with deque
from collections import deque
queue = deque()
queue.append(1)
queue.popleft()
```

2. Using Linked List for Random Access

```python
# BAD: if you need frequent access by index
linked_list.get(100)  # O(n) traversal

# GOOD: use array
array[100]  # O(1)
```

3. Not Considering Cache Performance

Summary

Key Takeaways

Arrays

- O(1) access, cache-friendly, but O(n) insert/delete in middle
- Dynamic arrays: O(1) amortized append via doubling strategy
- Use when: need random access, size known/stable

Linked Lists

- O(1) insert/delete at known positions, dynamic size
- Singly: 1 pointer, forward only; Doubly: 2 pointers, bidirectional
- Use when: frequent insertions/deletions, no random access needed

Stacks & Queues

- Stack (LIFO): function calls, undo, expression parsing, DFS
- Queue (FIFO): task scheduling, BFS, fair processing
- Deque: flexible both ends, palindromes, sliding windows

Implementation Guidelines

From Scratch vs Libraries

Implement from scratch to learn:

- Understand internal mechanisms
- Practice pointer manipulation
- Learn complexity trade-offs

Use library implementations for production:

- Python: `list`, `collections.deque`, `heapq`
- C++: `std::vector`, `std::list`, `std::stack`, `std::queue`
- Java: `ArrayList`, `LinkedList`, `Stack`, `PriorityQueue`

Remember

- Choose based on most frequent operations
- Consider cache performance for large datasets

Thank You!

Questions?

"The right data structure makes the algorithm simple.
The wrong data structure makes it impossible."