METROPOLITAN TIRANA UNIVERSITY

Course: Data Structures and Algorithms

Fundamental Data Structures
Arrays and Linked Lists

Evis Plaku

Arrays vs. Linked Lists: The Eternal Struggle

Arrays are great…

until they aren't!

Why study arrays and linked lists

- Arrays store data in order for fast access
- Linked Lists handle dynamic data without fixed size
- Used in web applications, databases, operating systems, games, and so many more applications

They help **store** and **manage data** efficiently in programs and real-world applications

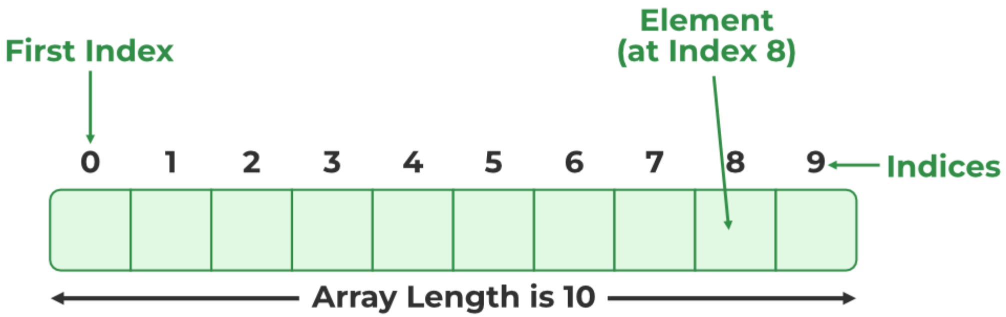
Array Fundamentals

What is an array: definition and properties

METROPOLITAN TIRANA UNIVERSITY

An array is a fixed-size collection of elements stored in continuous memory locations

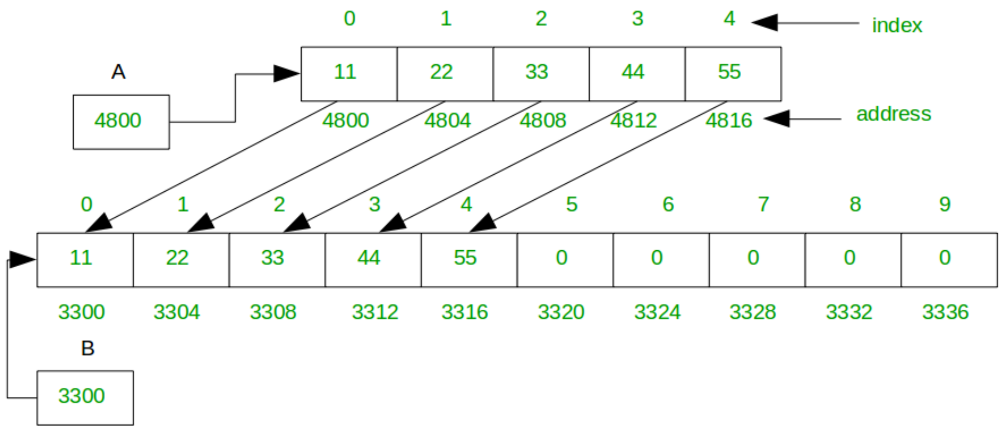
- Stores multiple values of the same type in order
- Elements are accessed using an index starting from 0
- Memory is allocated in a single, continuous block



Efficient for fast lookups but fixed in size

Static vs dynamic arrays

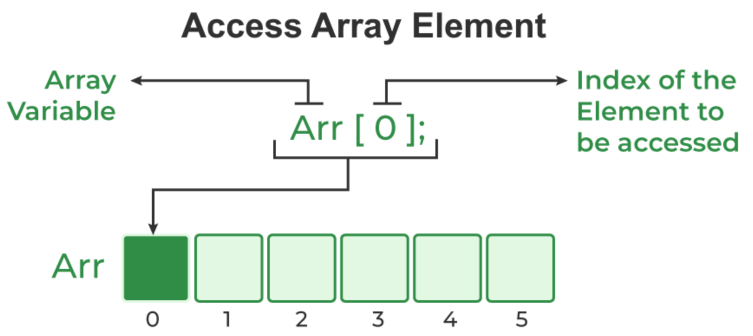
- Static arrays have a fixed size set at creation
- Dynamic arrays resize automatically when needed
- Static arrays are faster but waste memory if too large
- Dynamic arrays are flexible but need extra resizing time

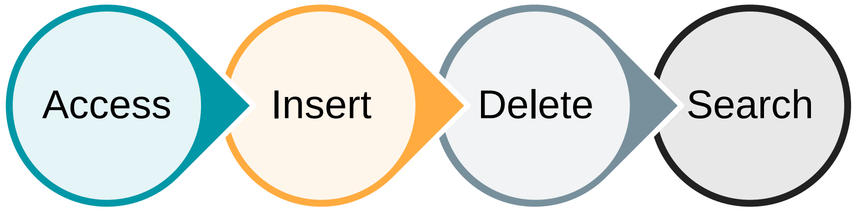


In Java, ArrayLists act as dynamic arrays by automatically resizing when needed

Array operations overview

- **Access**: retrieve any element instantly using its index
- **Insert:** add at index, may require shifting elements
- **Delete:** remove elements, shifting needed to fill the gap
- **Search:** finding an element using linear or binary search

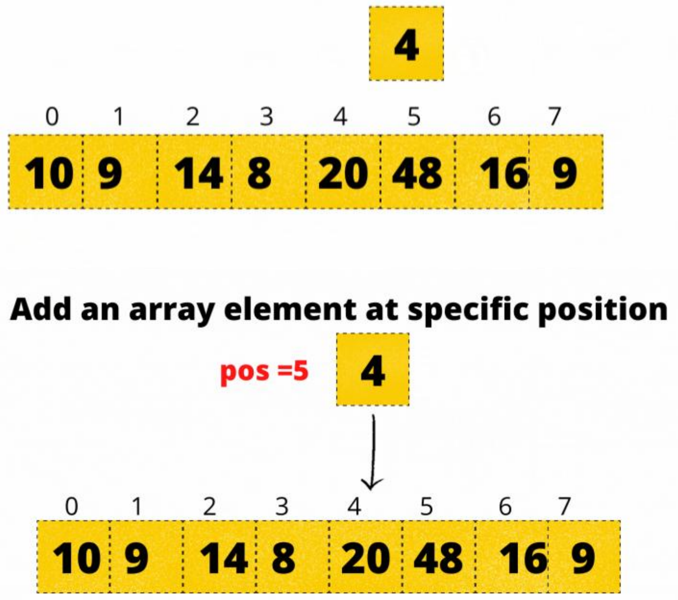




Insertion in arrays

Inserting elements in an array may involve shifting elements for space

- Insert at the end: O(1) time complexity, no shifts needed
- Insert at middle: O(n) time complexity, elements need shifting
- Shifting: Involves moving elements to create space for the new element



[Click to see interactively how an element is inserted]()

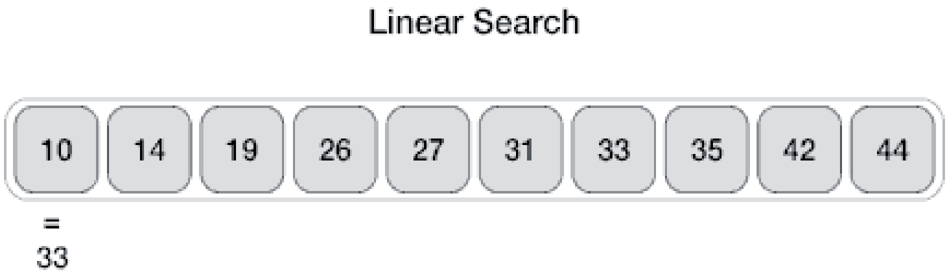
Search in arrays: linear search

Search for elements using linear or binary search based on array order

- **Linear search:** check each element sequentially until the target is found or array ends

- Simple but slow for large arrays

- $O(1)$ for best case; $O(n)$ for average and worst case



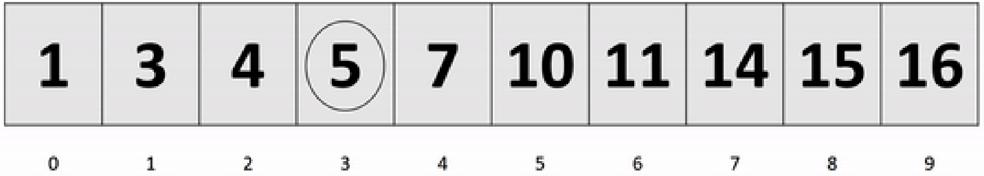
Search in arrays: binary search

Search for elements using linear or binary search based on array order

- **Binary search:** divides a <u>sorted array</u> in half, repeatedly narrowing down search range

- Faster but requires sorted data

- $O(1)$ for best case; $O(\log n)$ for average and worst case



Time and space complexity of arrays

METROPOLITAN TIRANA UNIVERSITY

Arrays offer **fast access** but have limitations in insertion, deletion, and resizing

Operation	Complexity
Access	$O(1)$
Insert (end)	$O(1)$
Insert (middle)	$O(n)$
Delete (end)	$O(1)$
Delete (middle)	$O(n)$
Linear search	$O(n)$
Binary search	$O(\log n)$

Use arrays for fast access and fixed-size data, but avoid frequent manipulations

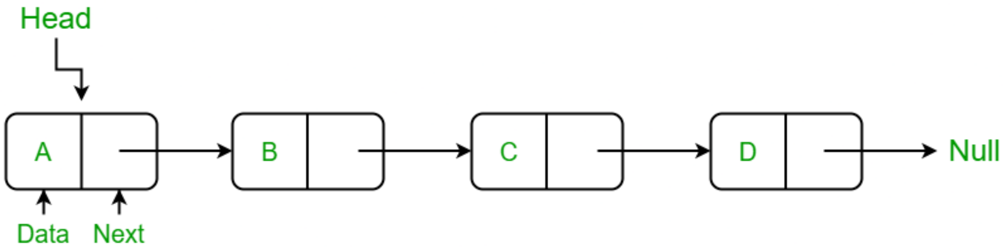
Linked List Fundamentals

What is a list: concept and node structure

METROPOLITAN TIRANA UNIVERSITY

A linked list is a collection of nodes, where each node points to the next

- Nodes contain data and a reference to the next node
- Head points to the first node, last node points to null
- Dynamic size allows efficient insertions and deletions



Used when data is frequently modified or added/removed

Singly linked list (SLL)

- A singly linked list has nodes that point only to the next node

```
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

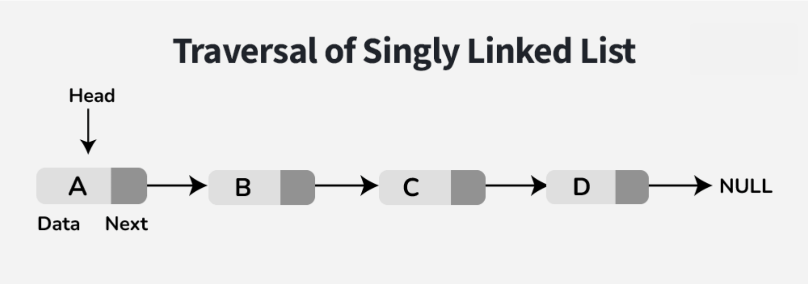
```
class SinglyLinkedList {
    Node head;

    SinglyLinkedList() {
        head = null;
    }

    void add(int data) {
        // To be implemented
    }

    void remove(int data) {
        // To be implemented
    }

    void display() {
      // To be implemented
    }
}
```

Key operations in lists: traversal

Traversal involves visiting each node from head to tail to access all data

- Start at the head and visit each node sequentially
- Continue until the last node (**next** is null)
- Used for printing or searching data in the list

```java
void traverse() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
}
```



Key operations in lists: insertion

METROPOLITAN TIRANA UNIVERSITY

Insertion can be done at the beginning, middle, or end of the list

- Insert at the beginning: new node becomes head
- Insert at the end: traverse and add after the last node
- Insert in the middle: traverse to the position and link the new node

```
void insertAtEnd(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
    } else {
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
}
```

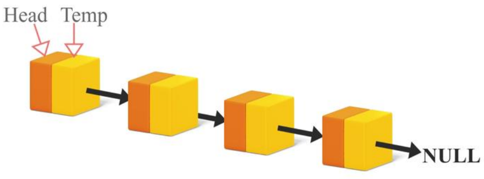
[Click here for an interactive visualization]

Key operations in lists: deletion

Deletion can be done at the beginning, middle, or end of the list

- Delete the first node: move the head to the next node
- Delete the last node: traverse to the second-to-last node and unlink
- Delete a specific node: traverse to the node before and unlink it

```
void deleteFirst() {
    if (head != null) {
        head = head.next;
    }
}
```

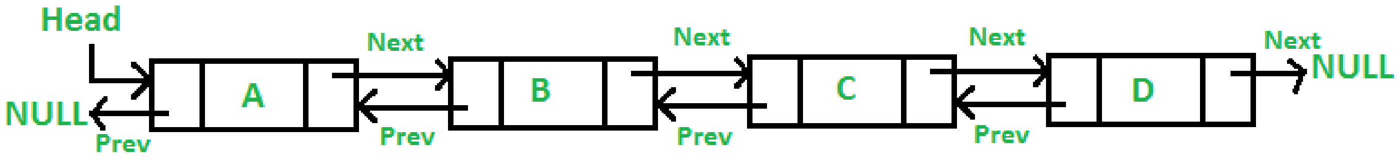


Never change or remove the head without ensuring the list stays intact

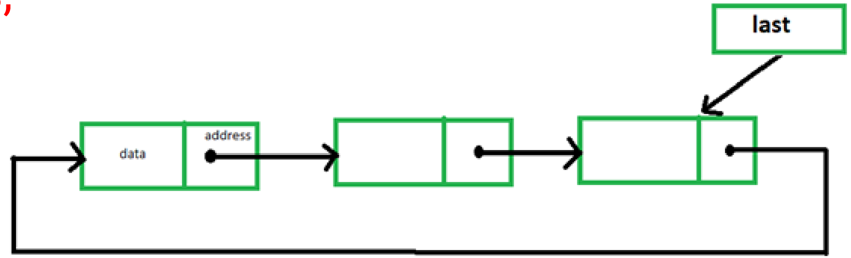
[Click here for an interactive visualization]

Other types of lists: doubly and circular

Doubly and circular linked lists offer more flexibility for complex operations



Singly linked lists are like one-way streets; sometimes you need a U-turn!

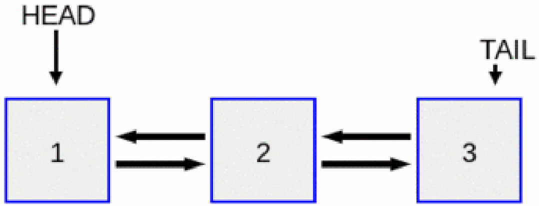


Doubly linked lists (DLL)

METROPOLITAN TIRANA UNIVERSITY

DLLs allow navigation in both directions, making insertions and deletions more flexible

- Each node has two references: `next` and `prev`
- Traversal from head to tail and tail to head
- Used in complex data structures like deques and browser history



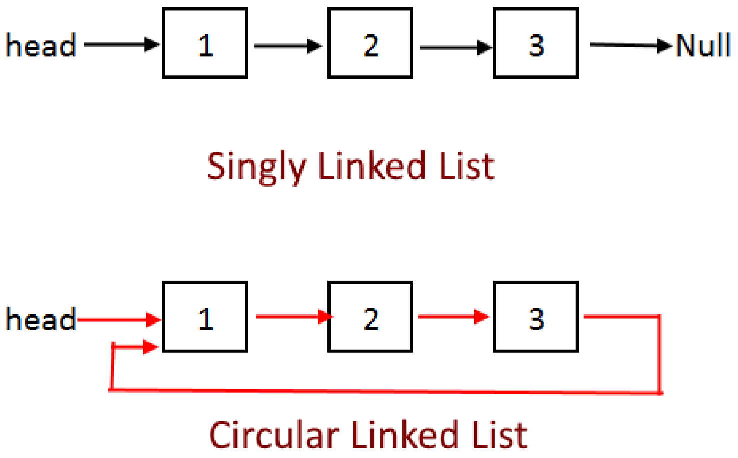
[Click here for an interactive visualization]

Circular linked lists (CLL)

METROPOLITAN TIRANA UNIVERSITY

CLLs have a circular structure, making them ideal for continuous cycles and buffers

- Tail node points to the head, forming a loop
- Traversal starts from head and loops back after reaching the tail
- Used in applications needing repetitive cycles (e.g., buffers, games)



Click here for an interactive visualization

Key Characteristics and Applications of Linked List Types

METROPOLITAN TIRANA UNIVERSITY

Singly Linked Lists

- One-direction traversal, simple structure
- Basic data storage, simple queues, and stacks
- Simple operations, space efficiency, and low overhead

Doubly Linked Lists

- Two-way traversal, more complex node structure
- Deques, browser history, undo functionality
- Bidirectional navigation, efficient insertions/deletions at both ends

Circular Linked Lists

- Circular structure, continuous looping
- Circular queues, round-robin scheduling, buffers
- Continuous cycles and circular data management

Complexity of key operations in lists

- Head operations: Insertion and deletion at the head are always $O(1)$

- Tail/Middle operations: Insertion and deletion from the tail or middle require $O(n)$ due to linear traversal

- Search: Searching requires $O(n)$ as each node must be visited one by one

Operation	Complexity
Insertion (at head)	$O(1)$
Insertion (at tail/middle)	$O(n)$
Deletion (from head)	$O(1)$
Deletion (from tail/middle)	$O(n)$
Search	$O(n)$

Understanding complexity helps optimize performance and resource usage

Real world applications of linked lists

Web browsers

Manage the history of visited pages going back and forth

Music players

Handle playlists to add and remove files

Undo functionality

Maintain and manipulate history of user actions

Polynomial arithmetic

Represent polynomials and conduct operations efficiently

Graph representations

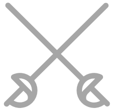
Used as helper data structures to build graphs

Arrays versus Linked Lists

Arrays versus linked lists: the overgoing battle

Feature	Arrays	Linked List
Memory allocation	Contiguous block, fixed size	Dynamic allocation, flexible size
Access time	O(1) for random access	O(n) for random access
Insertion/Deletion	O(n) (shifting needed)	O(1) at head/tail, O(n) elsewhere
Search	O(n) (sequential search)	O(n) (sequential search)
Memory efficiency	Inefficient for dynamic size	Efficient for dynamic size
Resizing	Fixed size (resize expensive)	Dynamic resizing (efficient)

Arrays are best when you need quick access and a fixed size



Linked lists work well for data that changes often, but access is slower and less efficient

Common mistakes and pitfalls

- **Assuming fixed size for arrays**: arrays cannot grow after creation; use ArrayList for dynamic sizing
- **Not handling null in linked lists**: always check for null pointers when traversing or modifying linked lists
- **Inefficient insertion in arrays**: inserting in the middle of an array requires shifting elements, which can be slow
- **Overusing linked lists for simple data**: linked lists are overkill when simple arrays or arraylists suffice

Key Takeaways

METROPOLITAN TIRANA UNIVERSITY

- Arrays and linked lists are fundamental data structures with distinct strengths and use cases

- Arrays provide fast access and efficient memory for fixed-size data

- Linked Lists are ideal for dynamic data that changes frequently

- Arrays are used in matrices, buffers, and lookup tables;
  Linked Lists in dynamic data like playlists, browser history,
  and undo functionality

Helpful resources on arrays and linked lists

- 🔍 VisualAlgo: an interactive platform that visualizes data structures and algorithms

- 🎥 BroCode video to learn linked lists in a concise and interactive manner

- 📚 Programiz: beginner-friendly tutorials on data structures and algorithms with clear explanations and examples

- 💻 FreeCodeCamp course on algorithms and data structures

Quote of the Week

METROPOLITAN TIRANA UNIVERSITY

Arrays provide speed in simplicity, but linked lists offer flexibility in complexity