

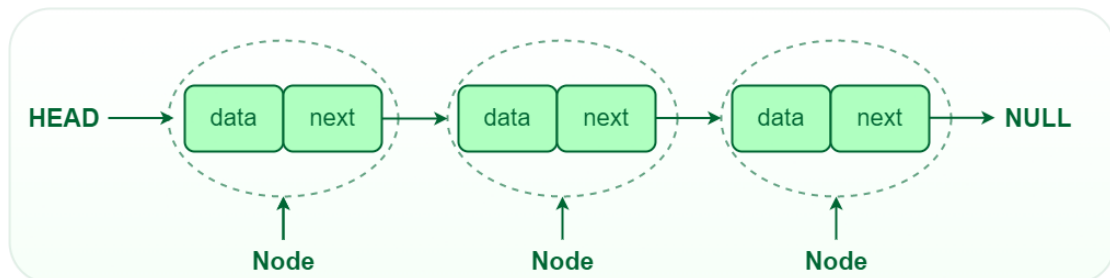


Islamic University of Gaza
Faculty of Engineering
Computer Engineering
Department



ECOM 3001 Data Structures and Algorithms Lab [Fall 2024]

Lab “02” Topic: Node and LinkedList



Prepared by:
Eng. Waseem Alfarram

Supervised by:
Dr. Ahmad Mahdi

Lab Objectives:

- To identify the advantages and disadvantages of using arrays.
- To think about another data structure that is similar to arrays and identify its advantages and disadvantages (**LinkedList**).
- To recognize what is **Node** and how to implement its class in java.
- To go deeper in what is LinkedList and what are its types (Singly, Doubly, Circularly).
- To implement the LinkedList types in java.
- To practice some exercises on LinkedList types.

Comparison between Arrays and LinkedLists:

Arrays:

Arrays Pros:

1. **Constant-Time Access by Index:** Accessing elements by index in an array is very fast and takes constant time (**$O(1)$**) because elements are stored in contiguous memory locations.
2. **Predictable Memory Allocation:** Arrays allocate memory in a single contiguous block, making it easier to predict and manage memory usage.
3. **Cache-Friendly:** Due to contiguous memory storage, arrays are more cache-friendly, which can lead to faster access times when dealing with large datasets.
4. **Useful for Fixed Size:** Arrays are suitable when the size of the data structure is fixed and known in advance.

Arrays Cons:

1. **Fixed Size:** Arrays have a fixed size, which means you need to know the maximum number of elements you will store in advance. This can lead to wasted memory if the size is overestimated or inability to accommodate more elements if underestimated.
2. **Inefficient Insertions/Deletions:** Inserting or deleting elements within an array can be inefficient, especially in the middle, as it may require shifting elements to maintain contiguous storage.

LinkedList:

LinkedList Pros:

1. **Dynamic Size:** Linked lists can dynamically grow or shrink, making them ideal for situations where the size of the data structure is not known in advance.
2. **Efficient Insertions/Deletions:** Insertions and deletions are efficient in linked lists since you can easily change references, without the need to shift elements as in arrays.
3. **No Wasted Memory:** Linked lists allocate memory for each element individually, avoiding wasted memory due to fixed sizing.
4. **Flexible Memory Allocation:** Linked lists allow for more flexible memory allocation, making it easier to manage memory in situations with varying memory requirements.

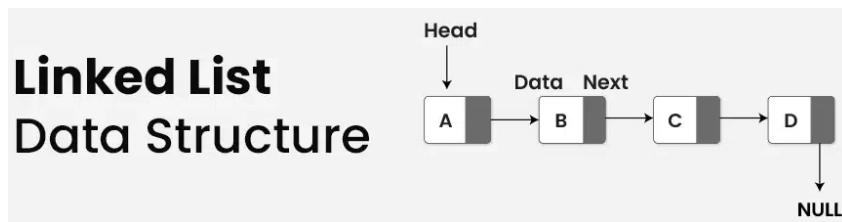
LinkedList Cons:

1. **Slower Access by Index:** Accessing elements in a linked list by index can be slower ($O(n)$) because you have to traverse the list from the beginning to reach the desired element.
2. **More Complex Implementation:** Linked lists are more complex to implement compared to arrays, as they involve managing references between nodes.
3. **Less Cache-Friendly:** Due to non-contiguous memory storage, linked lists can be less cache-friendly, potentially leading to slower access times for large datasets.

In general, the choice between arrays and linked lists depends on your specific use case and the operations you need to perform frequently. Arrays are great for fast index-based access with a fixed size, while linked lists are more flexible and efficient for dynamic sizing and efficient insertions/deletions.

Linked List Data Structure and Node Class:

What is a Linked List?



Linked List is a **linear data structure** which looks like a series of **nodes**, where each node has two parts: **data** and **next pointer**. Unlike Arrays, Linked List elements are not stored at a contiguous location. In the linked list there is a **head pointer**, which points to the first element of the linked list, and if the list is empty then it simply points to null or nothing.

Basic Terminologies of Linked List:

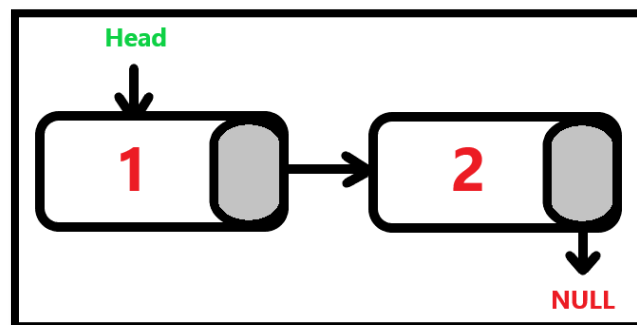
- **Head:** the Head of a linked list is a pointer to the first node or reference of the first node of linked list. This pointer marks the beginning of the linked list.
- **Node:** Linked List consists of a series of nodes where each node has two parts: data and next pointer.
- **Data:** is the part of node which stores the information in the linked list.
- **Next Pointer:** is the part of the node which points to the next node of the linked list.

Node Class Implementation:

```
static class Node <E> {  
    private E element;  
    private Node<E> next;  
  
    public Node(E e, Node<E> n) {  
        this.element = e;  
        this.next = n;  
    }  
  
    public E getElement() {  
        return element;  
    }  
  
    public Node<E> getNext() {  
        return next;  
    }  
  
    public void setNext(Node<E> next) {  
        this.next = next;  
    }  
}
```

Node Class Use:

```
public static void main(String[] args) {  
    // Creation of LinkedList from Node class: (1) -> (2)  
    Node newNode = new Node(1, new Node(2, null));  
  
    while (newNode.getNext() != null) {  
        System.out.print(newNode.getElement() + " -> ");  
        newNode = newNode.getNext();  
    }  
    System.out.println(newNode.getElement());  
}
```

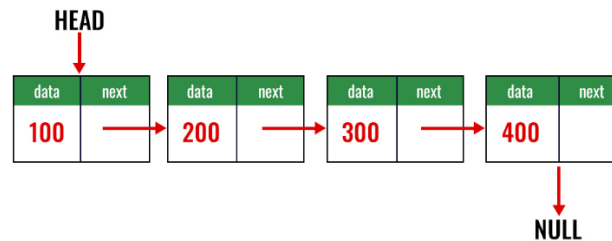


Types of Linked List Data Structure:

There are mainly three types of linked lists:

1. **Singly Linked List**
2. **Doubly Linked List**
3. **Circularly Linked List**

➤ Singly Linked List:



It is a type of linked list where each node has two parts: data and next pointer. The data part stores the information and the next pointer points to the next node of the linked list. The next pointer of the last node stores **null** as it is the last node of the linked list and there is no next node. Traversal of items can be done in the forward direction only due to the linking of every node to its next node.

Singly Linked List Class Implementation:

```
public class SinglyLinkedList {
    private Node<E> head;
    private Node<E> tail;
    private int size = 0;

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    Node<E> first() {
        return head;
    }

    Node<E> last() {
        return tail;
    }
}
```

Basic Operations on Singly Linked List:

- Insertion:
 - ✓ Inserting a node at the beginning of the list.
 - ✓ Inserting a node at the end of the list.
 - ✓ Inserting a node at a specific location in the list.
- Deletion:
 - ✓ Deleting a node from the beginning of the list.
 - ✓ Deleting a node from the end of the list.
 - ✓ Deleting a node at a specific location in the list.
- Traverse:

The process of displaying the elements of a Singly Linked list.
- Search:

The process of determining and retrieving a specific node either from the front, the end or anywhere in the list.

Implementation of Some Basic Operations on SLL:

- Inserting a node at the beginning of the SLL:

```
public void addFirst(Integer e) {  
    head = new Node(e, head);  
    if (isEmpty()) {  
        tail = head ;  
    }  
    size++;  
}
```

- Inserting a node at the end of the SLL:

```
public void addLast(Integer e) {  
    Node newest = new Node(e, null);  
    if (isEmpty()) {  
        head = newest;  
    }  
    else {  
        tail.setNext(newest);  
    }  
    tail = newest;  
    size++;  
}
```

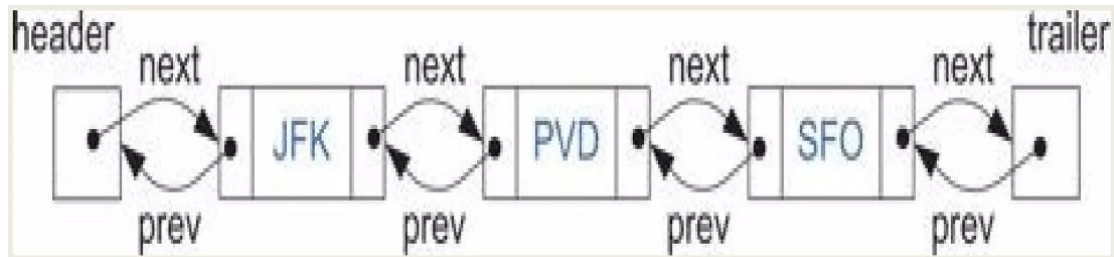
- Deleting a node from the beginning of the SLL:

```
public Integer removeFirst() {
    if (isEmpty()) {
        return null;
    }
    Node<Integer> removed = head;
    head = head.getNext();
    size--;
    // قبل ممكن يكون الهيد بيأشر عالتيل فلما أحذف العنصر لسا التيل محتفظ بقيمته فلازم أحذفها
    if (size == 0) {
        tail = null;
    }
    return removed.getElement();
}
```

- Traversing and displaying the elements of the SLL:

```
@Override
public String toString() {
    if (size == 0) {
        return "";
    }
    StringBuilder s = new StringBuilder();
    Node walk = head;
    while (walk.getNext() != null) {
        s.append("(").append(walk.getElement()).append(" -> ");
        walk = walk.getNext();
    }
    s.append("(").append(walk.getElement()).append(")");
    return s.toString();
}
```

➤ Doubly Linked List:



It is a type of linked list where each node has three parts: data, next pointer and previous pointer. The data part stores the information, the next pointer points to the next node of the linked list and the previous pointer points to the previous node of the linked list. The next pointer of the last node and the previous pointer of the first node stores **null**. Traversal of items can be done in the forward direction as well as backward direction due to the linking of every node to its next node as well as the previous node.

Doubly Linked List Class Implementation:

```
public class DoublyLinkedList {
    private static class Node<E> {
        private E element;
        private Node<E> prev;
        private Node<E> next;
        public Node(E e, Node<E> p, Node<E> n) {
            element = e;
            prev = p;
            next = n;
        }
        public E getElement() {
            return element;
        }
        public Node<E> getPrev() {
            return prev;
        }
        public Node<E> getNext() {
            return next;
        }
        public void setPrev(Node<E> p) {
            prev = p;
        }
        public void setNext(Node<E> n) {
            next = n;
        }
    }
}
```

```

private Node<E> header;
private Node<E> trailer;
private int size = 0;

public DoublyLinkedList() {
    header = new Node<>(null, null, null);
    trailer = new Node<>(null, header, null);
    header.setNext(trailer);
}

public int size() {
    return size;
}

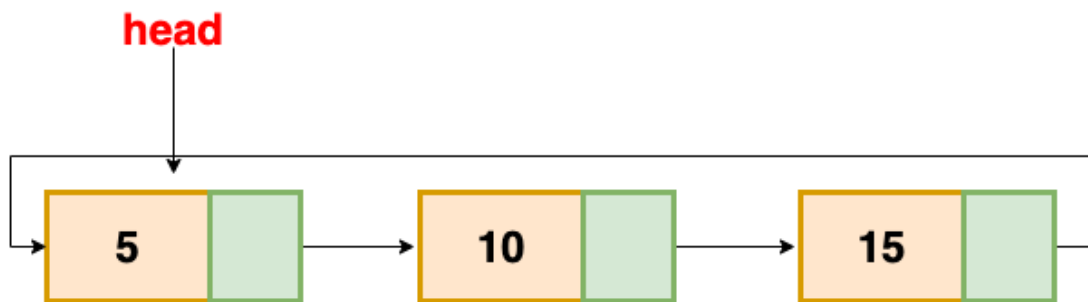
public boolean isEmpty() {
    return size == 0;
}

public E first() {
    if (isEmpty()) {
        return null;
    }
    return header.getNext().getElement();
}

public E last() {
    if (isEmpty()) {
        return null;
    }
    return trailer.getPrev().getElement();
}
}

```

➤ Circularly Linked List:



It is a type of linked list in which the first and the last nodes are also connected to each other to form a circle, there is **no NULL** at the end.

Circularly Linked List Class Implementation:

```
public class CircularlyLinkedList {
    private static class Node<E> {
        private E element;
        private Node<E> next;

        public Node(E e, Node<E> n) {
            this.element = e;
            this.next = n;
        }

        public E getElement() {
            return element;
        }

        public Node<E> getNext() {
            return next;
        }

        public void setNext(Node next) {
            this.next = next;
        }
    }

    private Node<E> tail = null;
    private int size = 0;

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return (size == 0);
    }
}
```

```
public E first() {  
    if (isEmpty()) {  
        return null;  
    }  
    return tail.getNext().getElement();  
}  
  
public E last() {  
    if (isEmpty()) {  
        return null;  
    }  
    return tail.getElement();  
}  
}
```

Exercises:

➤ Exercise 1: (Reverse Linked List)

Leetcode Question Link: <https://leetcode.com/problems/reverse-linked-list/>

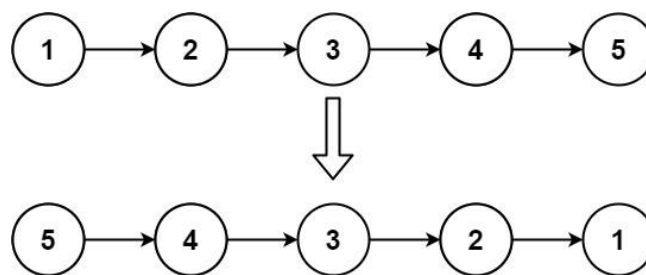
Given the head of a singly linked list, reverse the list, and return the reversed list.

```
public ListNode reverseList(ListNode head)
```

Example 1:

Input: head = [1, 2, 3, 4, 5]

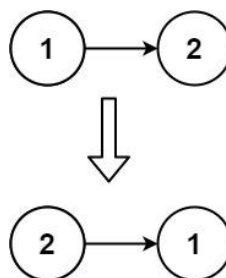
Output: [5, 4, 3, 2, 1]



Example 2:

Input: head = [1, 2]

Output: [2, 1]



Example 3:

Input: head = []

Output: []

Constraints:

- The number of nodes in the list is the range [0, 5000]
- $-5000 \leq \text{Node.val} \leq 5000$

➤ Exercise 2: (Merge Two Sorted Lists)

Leetcode Question Link: <https://leetcode.com/problems/merge-two-sorted-lists/>

You are given the heads of two sorted linked lists **list1** and **list2**.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

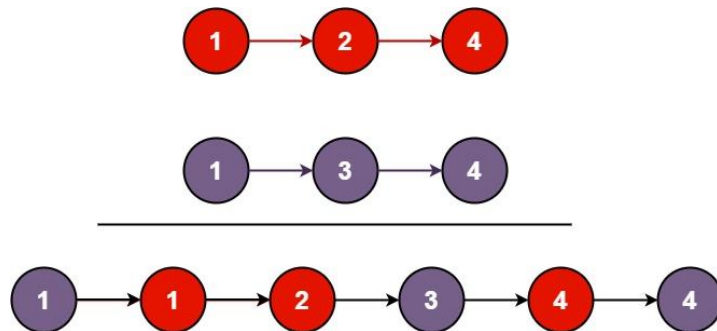
Return the head of the merged linked list.

```
public ListNode mergeTwoLists(ListNode list1, ListNode list2)
```

Example 1:

Input: list1 = [1, 2, 4], list2 = [1, 3, 4]

Output: [1, 1, 2, 3, 4, 4]



Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

Constraints:

- The number of nodes in both lists is in the range [0, 50].
- $-100 \leq \text{Node.val} \leq 100$
- Both **list1** and **list2** are sorted in **non-decreasing** order.

Bonus Leetcode Questions:

- <https://leetcode.com/problems/remove-duplicates-from-sorted-list/>
- <https://leetcode.com/problems/palindrome-linked-list/>
- <https://leetcode.com/problems/delete-nodes-from-linked-list-present-in-array/>

End of Lab 😊